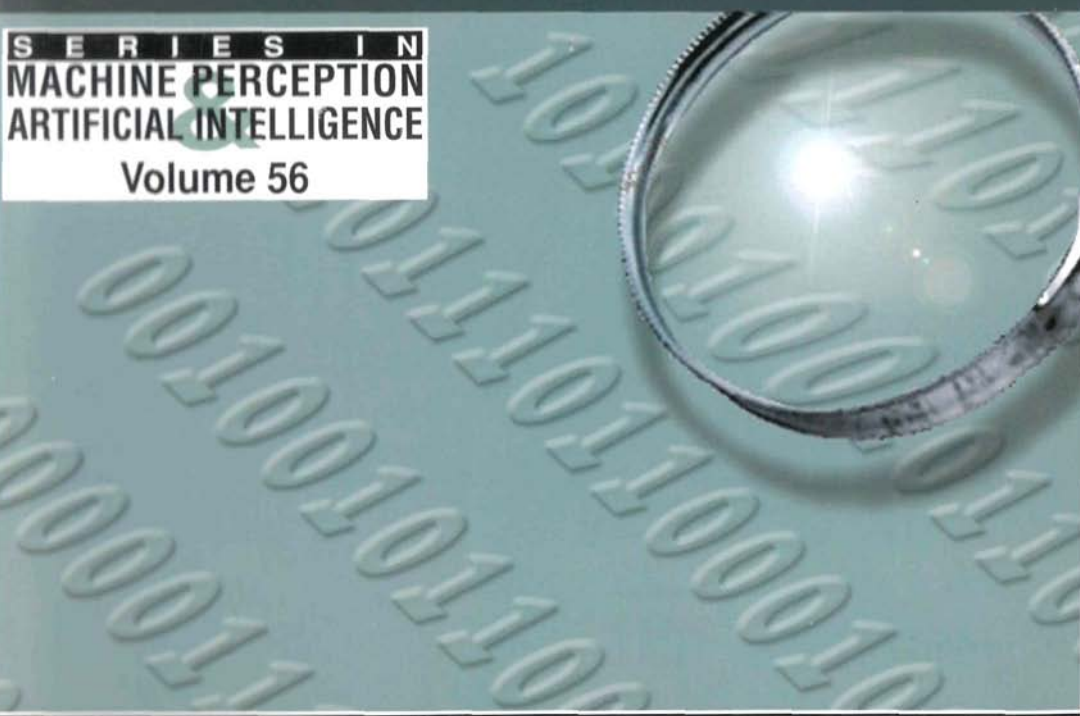
Artificial Intelligence Methods *in* Software Testing

Editors

Mark Last
Abraham Kandel
Horst Bunke

SERIES IN
MACHINE PERCEPTION
& ARTIFICIAL INTELLIGENCE
Volume 56



World Scientific

VISIT…

Artificial Intelligence Methods *in* Software Testing

SERIES IN MACHINE PERCEPTION AND ARTIFICIAL INTELLIGENCE*

Editors: **H. Bunke** (Univ. Bern, Switzerland)
P. S. P. Wang (Northeastern Univ., USA)

Vol. 43: Agent Engineering
(Eds. *Jiming Liu, Ning Zhong, Yuan Y. Tang and Patrick S. P. Wang*)

Vol. 44: Multispectral Image Processing and Pattern Recognition
(Eds. J. Shen, P. S. P. Wang and T. Zhang)

Vol. 45: Hidden Markov Models: Applications in Computer Vision
(Eds. *H. Bunke and T. Caelli*)

Vol. 46: Syntactic Pattern Recognition for Seismic Oil Exploration
(*K. Y. Huang*)

Vol. 47: Hybrid Methods in Pattern Recognition
(Eds. *H. Bunke and A. Kandel*)

Vol. 48: Multimodal Interface for Human-Machine Communications
(Eds. *P. C. Yuen, Y. Y. Tang and P. S. P. Wang*)

Vol. 49: Neural Networks and Systolic Array Design
(Eds. *D. Zhang and S. K. Pal*)

Vol. 50: Empirical Evaluation Methods in Computer Vision
(Eds. *H. I. Christensen and P. J. Phillips*)

Vol. 51: Automatic Diatom Identification
(Eds. *H. du Buf and M. M. Bayer*)

Vol. 52: Advances in Image Processing and Understanding
A Festschrift for Thomas S. Huwang
(Eds. *A. C. Bovik, C. W. Chen and D. Goldgof*)

Vol. 53: Soft Computing Approach to Pattern Recognition and Image Processing
(Eds. *A. Ghosh and S. K. Pal*)

Vol. 54: Fundamentals of Robotics — Linking Perception to Action
(*M. Xie*)

Vol. 55: Web Document Analysis: Challenges and Opportunities
(Eds. *A. Antonacopoulos and J. Hu*)

Vol. 56: Artificial Intelligence Methods in Software Testing
(Eds. *M. Last, A. Kandel and H. Bunke*)

Vol. 57: Data Mining in Time Series Databases
(Eds. *M. Last, A. Kandel and H. Bunke*)

Vol. 58: Computational Web Intelligence: Intelligent Technology for
Web Applications
(Eds. *Y. Zhang, A. Kandel, T. Y. Lin and Y. Yao*)

Vol. 59: Fuzzy Neural Network Theory and Applications
(*P. Liu and H. Li*)

*For the complete list of titles in this series, please write to the Publisher.

Series in Machine Perception and Artificial Intelligence – Vol. 56

Artificial Intelligence Methods *in* Software Testing

Editors

Mark Last
Ben-Gurion University of the Negev, Israel

Abraham Kandel
Tel-Aviv University, Israel
University of South Florida, Tampa, USA

Horst Bunke
University of Bern, Switzerland

World Scientific

NEW JERSEY • LONDON • SINGAPORE • BEIJING • SHANGHAI • HONG KONG • TAIPEI • CHENNAI

Published by

World Scientific Publishing Co. Pte. Ltd.

5 Toh Tuck Link, Singapore 596224

USA office: Suite 202, 1060 Main Street, River Edge, NJ 07661

UK office: 57 Shelton Street, Covent Garden, London WC2H 9HE

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library.

ARTIFICIAL INTELLIGENCE METHODS IN SOFTWARE TESTING
Series in Machine Perception and Artificial Intelligence (Vol. 56)

ISBN 981-238-854-0

Printed in Singapore by World Scientific Printers (S) Pte Ltd

Dedicated to

The Honorable Congressman C. W. Bill Young

House of Representatives

For his vision and continuous support in creating the National Institute for Systems Test and Productivity at the Computer Science and Engineering Department, University of South Florida

This page is intentionally left blank

Preface

Over the last 50 years, Artificial Intelligence (AI) researchers have developed a variety of computational models and methods that can make a computer program to behave in a reasonably "intelligent" way, i.e. to imitate such routine human tasks as learning, vision, speech recognition, etc. Numerous implementations of AI-based methods range from Optical Character Recognition (OCR) software to autonomous robots. Despite an apparent discrepancy between the way our brains think and the way computers work, the combination of computer power with human-like reasoning has generated impressive results in diverse domains. The purpose of this book is to shed the light on a relatively new and promising application area of AI techniques, namely *software testing*.

Software testing at various levels (unit, component, integration, etc.) has become a major activity in systems development. Though a significant amount of research has been concerned with formal proofs of code correctness, development teams still have to validate their increasingly complex products by executing a large number of experiments, which imitate the actual operating environment of the tested system. And, as we all know, the result is hardly satisfactory: commercial software products are abundant with undocumented "features", which are neither required nor expected by any rational customer. A failure to meet the stated requirements is usually caused by a design flaw, or a "bug". Discovering the hidden faults rather than proving that those do not exist in software is, in fact, the main purpose of software tests.

The state-of-the-art in software testing has a grave impact on the world's economy. Thus, leading software companies are forced to employ one tester per each active coder. Another industry trend is to transfer the testing activities to countries with cheap high-tech labor. A 2002 study by the US National Institute of Standards & Technology has found that "the national annual costs of an inadequate infrastructure for software testing is estimated to range from $22.2 to $59.5 billion" or about 0.6 percent of the US gross domestic product. This number does not include costs associated with catastrophic failures of mission-critical software such as the $165 million Mars Polar Lander shutdown in 1999 which was attributed to a software glitch. According to another report, U.S. Department of Defense alone loses over four billion dollars a year due to software failures.

The characteristics of software testing are quite similar to other problems successfully tackled by AI techniques. Thus, we need to find an optimal set of

actions (tests) subject to resource constraints (limited testing resources such as time) under uncertainty with respect to the number, severity, and location of undetected bugs. It is also important to understand the inherent probabilistic nature of software testing: we cannot require a test object to be "bug-free"; at most, we can assume that its likelihood to fail while in use is sufficiently low considering the criticality of the corresponding failure.

The authors of this book's chapters demonstrate the application of AI-based methods to various aspects of the software testing process as follows.

Chapter 1 by Witold Pedrycz and George Vukovich presents the fuzzy set-based approach to cause-effect software modeling as a basis for designing a test oracle in black-box software testing. Such an oracle can be used to predict the outcome of a given test.

A tested object can be executed an unlimited number of times thus generating large amounts of execution data. **Chapter 2** by Mark Last and Menahem Friedman demonstrate the potential use of data mining algorithms for automated induction of functional requirements from such data. In addition to being a test oracle, the resulting model can be utilized for designing a minimal set of test cases.

Data-driven techniques for automated testing are not applicable to Graphical User Interface (GUI) software, which is characterized by event-driven input and graphical output. A novel method for automated GUI regression testing using AI planning is presented by Atif M. Memon in **Chapter 3**. The proposed technique is based on re-planning GUI test cases by associating a task with each test case.

Artificial Neural Networks (ANN) is a well established method for learning complex functions from training examples. **Chapter 4** by Prachi Saraph, Mark Last, and Abraham Kandel shows the use of neural networks for automated input-output analysis of data-driven programs. The proposed methodology is based on advanced algorithms for efficient network pruning.

Effectiveness of software testing can be increased dramatically if one can estimate the quality of each system module prior to test design and execution. In **Chapter 5,** Taghi M. Khoshgoftaar and Naeem Seliya present the three-group classification model for predicting the number of faults as a function of common software metrics.

The distribution of faults across system modules may be *skewed* meaning that only a few complex modules are prone to failure. **Chapter 6** by Scott Dick and Abraham Kandel uses the machine learning technique of resampling to overcome the problem of skewness in software metrics databases. The resampling methodology can improve the performance of any data mining algorithm applied to software testing data.

We believe that the chapters collected in this book present only a small, but representative sample of potential AI applications in the extremely important

areas of software testing and software quality assurance. As more researchers will contribute to this challenging field, we expect to see evolvement of practical AI-based tools for automated testing of complex software systems.

Acknowledgments

We thank all the contributing authors who, despite their busy schedule, have responded enthusiastically to our invitation and have done such an excellent job on their chapters. We would like to thank Ronit Levy who has done final formatting of all chapters. Our special thanks go to Editor Ian Seldrup for his prompt and patient attitude. The preparation of this volume was partially supported by the National Institute for Systems Test and Productivity at the University of South Florida under U.S. Space and Naval Warfare Systems Command grant number N00039-01-1-2248. We would also like to acknowledge the partial support provided by the Fulbright Foundation to Prof. Kandel. During the academic year 2003-2004 Prof. Kandel has been awarded Fulbright Research Award at Tel-Aviv University, College of Engineering.

September 2003

Mark Last
Abraham Kandel
Horst Bunke

This page is intentionally left blank

Contents

CHAPTER 1

FUZZY CAUSE – EFFECT MODELS OF SOFTWARE TESTING

Witold Pedrycz

Dept. of Electrical and Computer Engineering
University of Alberta, Edmonton, Canada T6G 2G7
and
Systems Research Institute, Polish Academy of Sciences
01-447 Warsaw, Poland
E-mail : pedrycz@ee.ualberta.ca

George Vukovich

Canadian Space Agency,
Spacecraft Engineering St. Hubert, Quebec J3Y 8Y9 Canada
E-mail: George.Vukovich@space.gc.ca

Software testing is an important and challenging endeavor. In this study, we concentrate on the fuzzy set-based model of cause-effect software testing being regarded as a generalization of the existing standard technique of black box testing. The essence of this approach lies in the formation of a network of logic relationships between causes and effects expressed by means of logic *and* and *or* operations as well as fuzzy predicates. The underlying architectural framework proposed in this study involves two general classes of processing units, namely fuzzy constraint predicates and logic units. The first category of these constructs generalizes the well known Boolean predicates of equality (equal to) and inclusion (less or equal to) that are now articulated in the language of fuzzy logic. More specifically, the Boolean equality predicate becomes generalized to the form of the multivalued (fuzzy) equality operation while the inclusion operation captures a multivalued implication operation. We show how these operations are used to construct a multivalued generalization of the standard and well-known cause-effect

models of testing. We elaborate on the direct problem, that is a design of the cause-effect model based on experimental data and then concentrate in detail on the development of testing oracles. This process is referred to as an inverse problem. The solution to this problem is formulated as a certain gradient – based learning task. Moreover we show how the software requirements could be refined (quantified) through the learning. The study discusses a detailed scheme of parametric optimization of the inputs (causes) being a part of the required testing oracles

1. Introduction

Logic is the underlying fabric of all software constructs. It is the logic that permeates the requirements, architectural considerations, design, and testing. The continuously growing complexity of software products and processes calls for comprehensive and thorough testing techniques cf. Perry [6] as well as Weyuker, and Goradia [11]. Usually these techniques are classified into two main categories that is black box and white box testing, refer to Meyers [1], Perry [6], and Shooman [8]. With the emergence of Computational Intelligence, its breadth of methodological and computational pursuits were investigated and exploited in the realm of software testing. Some initial findings look promising and encourage further research down this path. These approaches dwell on the concepts and algorithmic underpinnings of neural networks, fuzzy sets and evolutionary computing, Pedrycz and Peters [5] , von Mayrhauser et al. [9], and Vanmali, Last, Kandel [10]. The logic fabric of software makes the use of fuzzy sets particularly attractive in the sense that it implies logic-based models of testing to be of particular relevance. As a matter of fact, cause-effect models of testing, see Perry [6], being commonly used in black box testing dwell directly upon the logic type of relationships between causes and effects. Let us recall that they help test a system vis-à-vis software specifications. As the requirements state what effects are expected for specific causes, building effective testing oracles (that is pairs of causes and desired effects) becomes feasible. In a nutshell, cause-effect testing is based upon some cause-effects graphs, see Figure 1. These graphs are formed by using *and* and *or* operations (that are located at the nodes of the graph) with some links associated with a

negation operation (*NOT*). What is shown in Figure 1, is just a simple graph concerning a toy ATM banking transaction system having a few functions (credit, debit, account number valid, etc) that are causes in our model and several ensuing effects (such as invalid account number, debit account, credit account, etc).

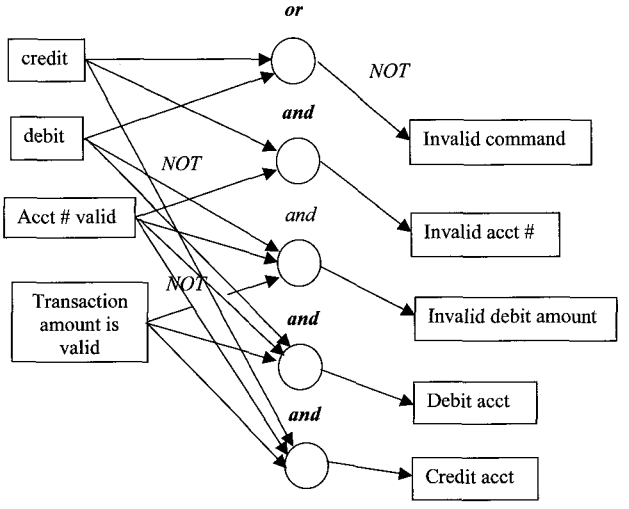


Figure 1. A simple cause-effect graph

On the basis of such graph, we construct a collection of testing (test) oracles that is a pair of causes-effects in which we exercise only a single effect (only a single output node of the graph becomes "active"). These oracles are a key mechanism to test a system: we apply a cause vector (being a part of the given oracle) and compare the output of the system versus the associated target effect vector (being a part of the original collection of the requirements).

Apparently, in the above model the causes and effects are binary (Boolean, yes-no). There are, however, a number of systems where we encounter continuous causes and effects. Consider a situation where the specifications are formed as illustrated in Figure 2.

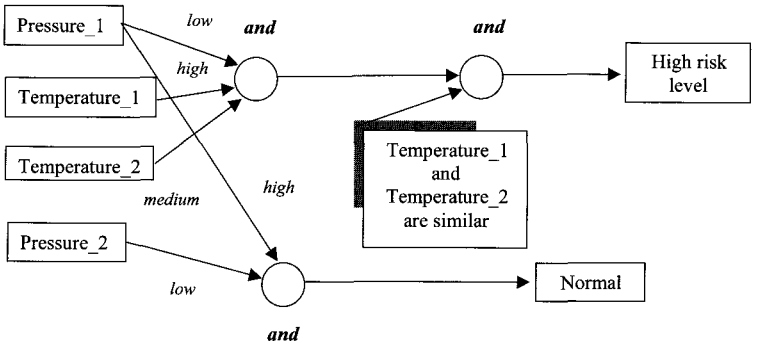


Figure 2. A simple cause-effect graph concerning a fragment of software for controlling a certain industrial installation. Note that the variables of pressure and temperature are continuous; furthermore some causes are related (shown in a shadowed box)

As we are concerned with continuous variables of pressure and temperature, their quantification is inherently continuous and can be realized in the language of linguistic terms, so are the effects (high risk level and normal operation). Again the graph exhibits a strong logic flavor yet its formalization in the language of two-valued logic may result in undesired brittleness phenomena. The problems of this nature motivate further refinement of the original binary cause-effect graphs and pose issues regarding their ensuing testing. The objective of this study is to introduce a fuzzy set based model of the cause-effect testing scheme. We propose a generalization of the graph in the sense of generalized logic operators and constraints (fuzzy predicates) to be included in this construct. In this setting, we revisit the task of building testing oracles and reformulate it as an inverse problem in such fuzzy graphs.

In the study, we confine ourselves to all variables taking on values in [0,1] and exploit a formal structure of fuzzy sets. In particular, all logic operations (*and* and *or*) are realized in the language of triangular norms and co-norms as commonly encountered in the technology of fuzzy sets. Subsequently, other operations used here such as similarity (matching) and inclusion originate are rooted in the realm of fuzzy sets.

2. Architectural considerations

We start with a discussion on the two categories of elements in the graphs, namely constraint predicates and basic logic nodes (fuzzy AND and OR neurons).

2.1. General constraint predicates

We consider the following binary (viz. two-argument) predicates, namely equality (=) and inclusion (less or equal to, <=). The third one, dominance (greater than or equal to, =>) is the dual to the inclusion predicate. These are obvious two-variable relations. The generalization is straightforward as we relax the predicates by introducing the following fuzzy set counterparts of the similarity and inclusion operations, cf Pedrycz and Gomide [2], Pedrycz [3,4]. More formally, we say that the degree of similarity (equality) of "*a*" and "*b*" is given as

$$a \equiv b = (a \rightarrow b)(b \rightarrow a)$$

and returns a value in [0,1] where the implication operation is a realized as a residuation operator Pedrycz [4] defined in the following manner

$$a \rightarrow b = \sup\{c \in [0,1] | atc \leq b\}$$

while "*t*" denotes a certain *t*-norm. Note that the equality operation returns 1 if and only if when both arguments are identical. The larger the difference between "*a*" and "*b*", the lower the value of the corresponding degree of similarity. The schematics of the operation and its input-output characteristics are visualized in Figure 3. Alluding to the nature of the processing realized there, we will be referring to this realization of the predicate as a matching (equality) neuron.

The processing realized by the inclusion operation (inclusion neuron) is based on the inclusion operation (implication). Note that if a <b then the inclusion returns 1 (which simply means that the degree of inclusion of "*a*" in "*b*" is one); this result is highly intuitive. In the opposite case, that is $a > b$, we have a monotonic decline in the value of the output of this neuron: the higher the value of "*a*" versus "*b*", the lower the value of

this operation. Figure 4 shows the symbol of the inclusion neuron, *incl(a,b)* along with its characteristics. The graph there concerns the residuation operation induced by the product operation.

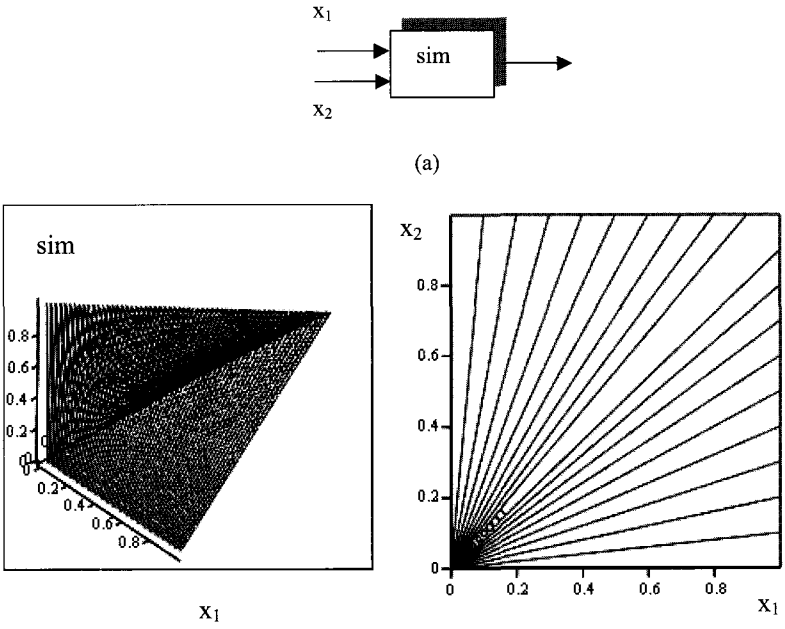


Figure 3. The equality (similarity) neuron: symbol (a) and its 3D and contour plot characteristics (b) (residuation is induced here by the product operation, $a \rightarrow b = min\ (1, b/a)$)

More specifically, this gives rise to the expression:

$$a \rightarrow b = \begin{cases} 1 \text{ if } a \leq b \\ a/b \text{ if } a > b \end{cases}$$

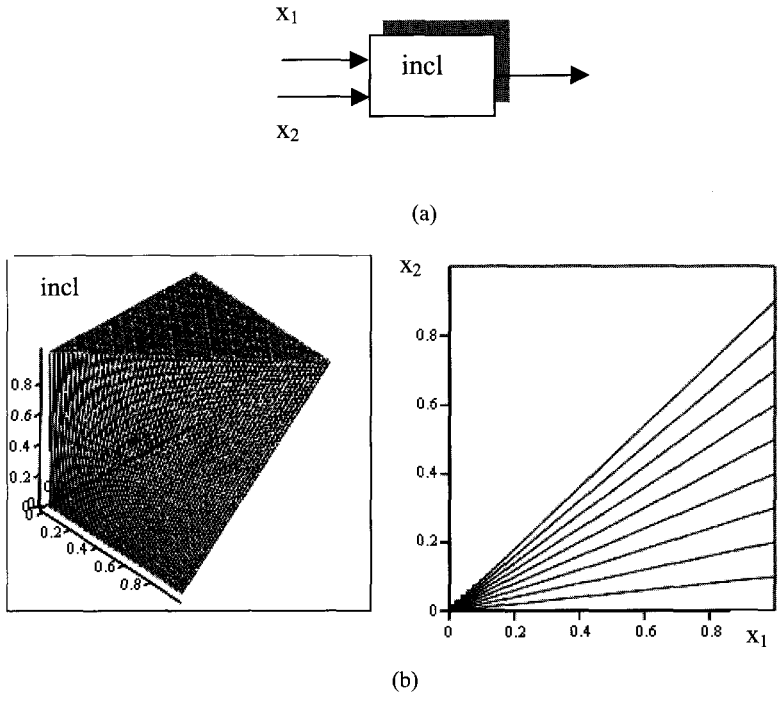


Figure 4. The inclusion neuron: symbol (a) and its characteristics (b) (residuation induced by the product operation)

The dominance operation *dom (a, b)* is the inclusion operation with the reversed order of the arguments.

2.2. The basic logic processing unit

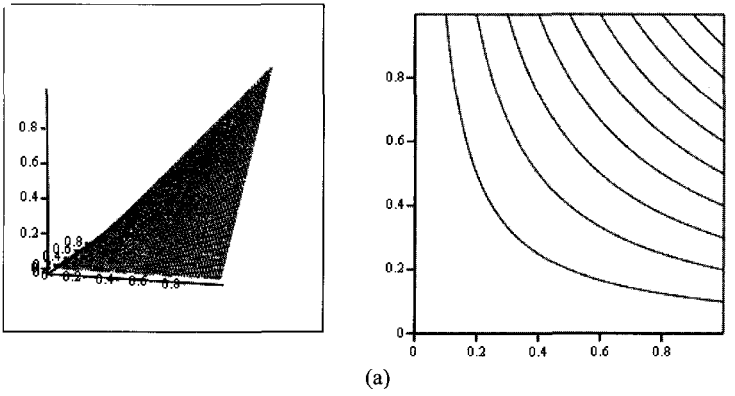
We consider two types of basic logic processing units (logic neurons), namely, AND and OR neuron. Proceeding with the computational details we have, cf. Pedrycz [4]. An AND neuron maps n-inputs $\boldsymbol{x}$ in the unit hypercube to a single output, y = AND ($\boldsymbol{x}$; $\boldsymbol{w}$) where $\boldsymbol{x}$ and the weights (connections) $\boldsymbol{w}$ are combined using a standard t-s convolution

$$y = \underset{i=1}{\overset{n}{S}}(x_i t w_i) \tag{1}$$

More specifically, the inputs are combined with the corresponding connections using some s-norm and afterwards all these partial results are aggregated and-wise (using a certain t-norm).
An OR neuron, y = OR ($\boldsymbol{x}$; $\boldsymbol{w}$) carries out the following transformation (convolution)

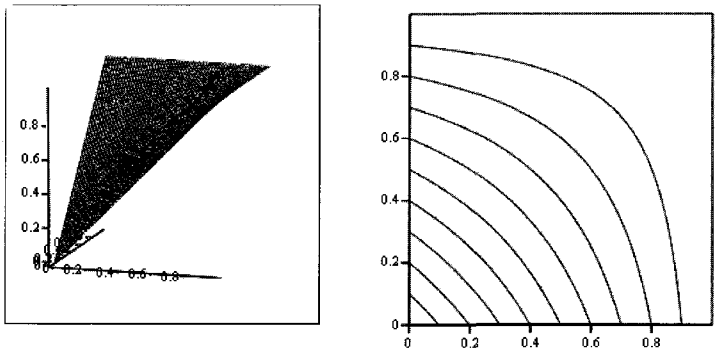
$$y = \underset{i=}{\overset{n}{T}}(x_i s w_i) \tag{2}$$

where the t- and s- norms are reversed in terms of their use. The role of the weights is to calibrate the impact of the individual variables on the output of the neuron. Considering the AND neuron and bearing in mind the boundary conditions of the triangular norms and co-norms, we note that the lower the values of the connections, the more essential the input of the neuron on its output. The reverse effect holds for the OR neuron where more significant impact of the input is associated with the higher value of the corresponding connection. The plots of the AND and OR neurons for the weight vector $\boldsymbol{w}$ equal to [0.0 0.0] for the AND neuron and $\boldsymbol{w}$ = [1.0 1.0] for the OR neuron are visualized in Figure 5. Note that these neurons generalize standard (binary) OR and AND logic gates, namely for the binary inputs these neurons coincide with the Boolean operations of conjunction and disjunction.



(a)

Figure 5. 3D plots of characteristics of logic neurons: AND (a) and OR (b) (t-norm: product, s-norm: probabilistic sum, $a + b - ab$)



(b)

Figure 5 (cont.). 3D plots of characteristics of logic neurons: AND (a) and OR (b) (*t*-norm: product, *s*-norm: probabilistic sum, $a + b - ab$)

3. The fuzzy cause – effect networks: mapping software specifications

The operations and processing units (fuzzy predicates and logic neurons) are assembled together in order to describe the set of requirements. They capture the essence of requirements by expressing them in the language of logic aggregates and relationships that reflect the character of the constraints (dependencies) occurring between such specific variables. In what follows, we provide a detailed insight into the problem by showing a number of illustrative examples.

3.1. Illustrative examples of the networks

A number of small architectures of the networks is shown here in order to gain a better grasp of the processing realized by the logic neurons and a role being played by the individual predicates (constraints).

A. The architecture in Figure 6 (a) describes the AND combination of x_1 and x_2 where x_1 and x_2 are assumed to be *similar* (where this predicate is modeled in the form of the equality neuron). The

connections of the AND neuron are equal to zero (so the inputs exhibit the same level of impact on the output of the neuron)

B. This case is similar to (A) but now we consider that x_1 is included in x_2 (or equivalently, x_2 dominates x_1). Note a difference between the characteristics of this neuron in comparison to the plain AND neuron (in which we did not incorporate any relational constraints)

C. This case is the same as (A) but now the inputs (causes) are aggregated OR-wise. The connections of the neuron are set to ones.

D. Here we have the structure (B) but with the OR neuron being used to aggregate the causes

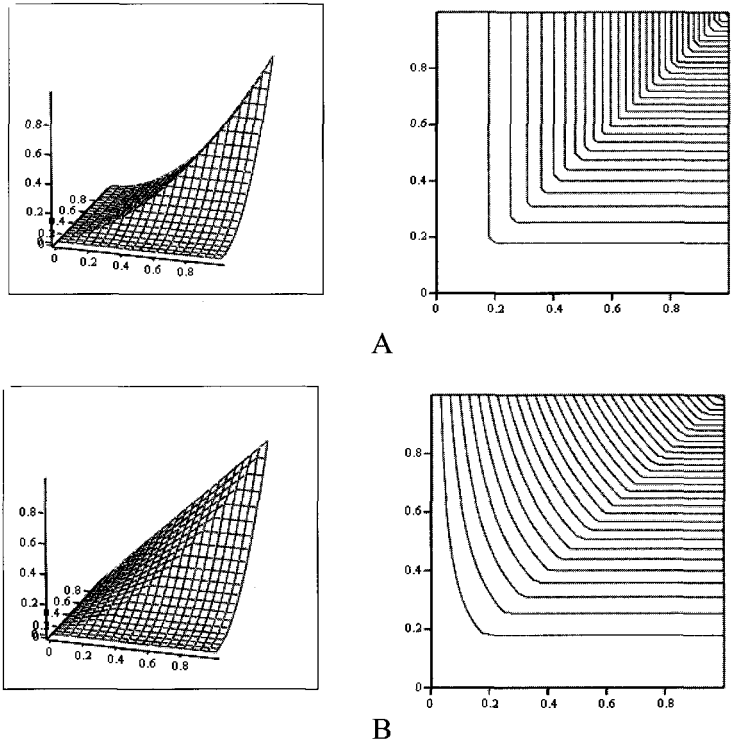


Figure 6. Logic aggregation of x_1 and x_2 with x_1 and x_2 being subject to some relational constraints; plots show 3D characteristics and contour plots; see a detailed description in the text

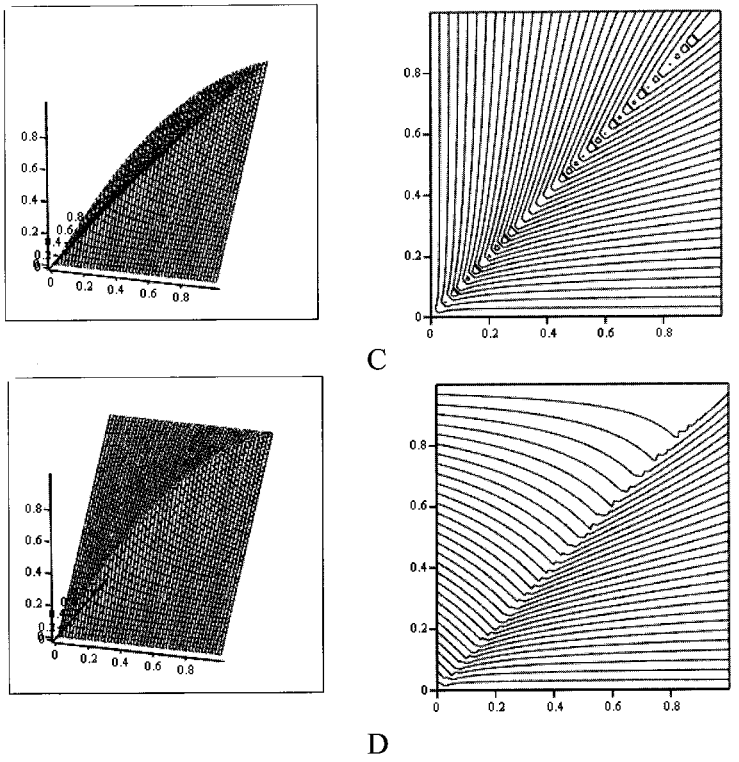


Figure 6 (cont.). Logic aggregation of x_1 and x_2 with x_1 and x_2 being subject to some relational constraints; plots show 3D characteristics and contour plots; see a detailed description in the text

We can conclude that the resulting networks are highly heterogeneous and lead to a number of interesting nonlinear characteristics.

In general, we anticipate a multiple input – multiple output cause effect network in which a number of causes are associated with effects. An example of such network is shown in Figure 7; here four causes are linked with two effects. There are also two predicates expressing the relationship between x_1 and x_2 and x_3 and x_4, respectively.

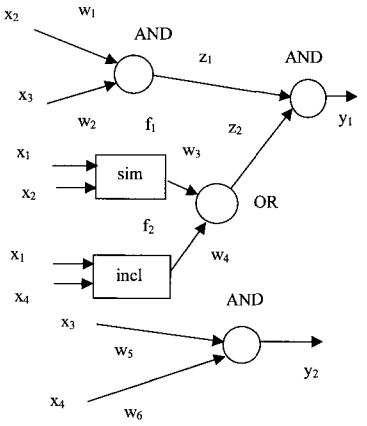


Figure 7. An example of the cause-effect network

We associate the network with a straightforward logic interpretation. As a matter of fact, we can easily come up with an equivalent logic description of the network (note that this description is up to the values of the connections as their numeric entries are not a part of the logic fabric of the statement)

if x_2 *and* x_3
Subject to constraints:
x_1 is *similar* to x_2
or
x_3 is *included* in x_4
then cause y_1

if x_2 *and* x_4 then cause y_2

(the connections of the network are set up as follows:
$w_1 = 0.6$; $w_2 = 0.1$; $w_3 = 1.0$; $w_4 = 0.9$; $w_5 = 0.0$; $w_6 = 0.0$)

It is worth stating that the network shown above is an interesting example of a multilevel fuzzy relational equation, see [2][4]. In particular, we have: (a) OR neurons are fuzzy relational equation with the *s-t* composition operator, (b) AND neurons are fuzzy relational equations with the *t-s* composition operator (that are dual to the previous category of the equations), and (c) constraint predicates are examples of referential fuzzy relational equations. The multilevel nature of the equations make them difficult (if not impossible) to solve analytically. Subsequently we have to resort to optimization methods (iterative gradient-based techniques) when determining unknown fuzzy sets of causes or estimating the parameters of the network. In the setting of fuzzy relational equations, there are two fundamental classes of problems: (i) direct, and (ii) inverse task of optimization. The first problem is concerned with the determination of the parameters of the network (fuzzy relations) for given causes and effects. The second task deals with the determination of the inputs (independent variables) for the given outputs (that is dependent variables-effects). In the ensuing section, we show that these two tasks correspond to the two fundamental problems of requirement generation and building test oracles.

4. The construction of the network: a direct problem

The cause-effect logic network captures and structuralizes the knowledge about the requirements of the system. The direct problem identified in the previous section is about the generation and numeric quantification (refinement) of software requirements. There are two essential sources of information when dealing with the construction of the networks:

Qualitative domain knowledge. It is acquired through the process of converting some preliminary requirements into a series of logic descriptors, namely logic operators and relational constraints organized into a single network. This type of knowledge is predominantly qualitative as we develop (construct) a skeleton of the network that is conveniently arranged into a collection of the neurons

Numeric data sets are afterwards used to calibrate the network that is optimize the connections of the neurons located in this structure. This is accomplished through a process of parametric learning where we are

provided with a collection of input-output cases treated as a learning scenario. As the structure has been already fixed, the optimization is governed by a standard gradient-based update mechanism for the connections (weights) of the neurons.

The direct problem in the networks of this character is explicitly linked to the issue of requirement development in two main ways

- First, the requirements (pairs of associated causes and effects) can be easily "downloaded" onto the structure of the network. This is possible because of the flexibility of the structure of the network that fully reflects the qualitative skeleton of the relationships existing in the collection of the requirements. The nature of the processing being in full agreement with the predicates (constraints) and the logic nature of the processing encountered there facilitates this downloading.
- Second, once the parametric training (adjustments) of the parameters of the network proceeds on a basis of some input-output pairs, the requirements could be checked for consistency as we can compare the results produced by the network versus the given requirements. Potentially inconsistent specifications can be flagged and afterwards carefully revisited.

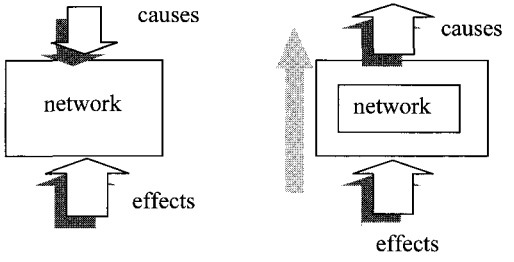


Figure 8. Direct (a) and inverse (b) problem and the experimental data being used in the calibration of the network

The crux of the direct problem (estimation of the parameters of the network) is portrayed in Figure 8; the collection of the input-output pairs can be treated as a numeric manifestation of the general qualitative requirements. The essence of the inverse problem, as visualized in

Figure 8(b), is to form a vector of causes that correspond to the given effects. The solution may not be unique and this is usually the case; for the same effect we may come up with several causes. We say that there is an equivalence class of causes implied by a certain effect.

5. An inverse problem: forming a mechanism of generating testing oracles

The network introduced in the previous section is a means to capture and calibrate the specifications. We have demonstrated that the network could be learned meaning that its parameters can be adjusted on a basis of some learning set. Generating testing oracles can be done by solving an inverse problem in which for a fixed collection of effects, we produce the corresponding collection of causes. More descriptively, the determination of the testing oracles – causes is carried out by moving backward from the output of the network to its inputs; hence the origin of such inverse problem.

The inverse problem concentrates exclusively on the parametric learning as we determine the input vector (causes) for a fully connected structure and parameters of the network. As such, a standard gradient-based learning (optimization scheme) is appropriate. To discuss the main steps of the solution, we continue with an example in Section 3. Let us assume that the required vector of effects is [1 0 0 0] meaning that we are looking for a cause **x** that lead to specific binary effect (the first output of the network, effect_1 is present). The gradient-based learning assumes the well-known form

$$\boldsymbol{x}(\text{new}) = \boldsymbol{x} - \alpha \nabla_{x} Q \tag{3}$$

where $\nabla_{x} Q$ is the gradient of the performance index (Q) with respect to the inputs (causes). The learning rate (α) assumes positive values and controls the pace of changes (updates) of the values of the causes. The performance index Q reflects a difference between the desired output of the network and the actual one that results from a certain input (causes). Commonly, this difference is expressed as a distance between these two vectors. The Euclidean distance serves a standard vehicle to quantify this effect. As the structure of the network is given, we use it in the detailed

calculations of the gradient. Alluding to the structure shown in Figure 7, we carry out detailed computations (assuming the Euclidean form of the distance function between the target and the output of the network),

$$Q = (target_1 - y_1)^2 + (target_2 - y_2)^2 \tag{4}$$

The derivations of the detailed learning scheme are not so apparent as the network is highly heterogeneous (the logic neurons and predicate neurons are scattered across the entire architecture). In a nutshell, the well known scheme of backpropagation learning needs to be carefully revisited. Table 1 contains the details of the computing sequence which follows a way in which the error backpropagates across the network. In general, we start with the output processing units and moving to the inputs of the network:

As there are two outputs, the inverse problem gives rise to the following two testing oracles. For the first one we consider the target vector to be equal to [1 0]. The second oracle concerns the target being in the form of the binary vector equal to [0 1]. The learning completed for these two targets is quantified in terms of the performance index Q obtained is successive learning epochs, see Figure 9.

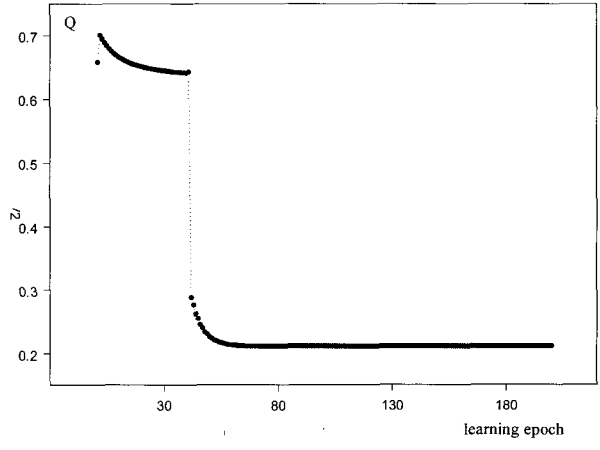


Figure 9. Performance index in successive learning epochs for the first testing oracle; learning rate α set to 0.05

Table 1. The detailed expressions for learning realized in the cause-effect network; refer to Figure 7 for the detailed notation (s-norm: probabilistic sum, t-norm: product)

$$\frac{\partial Q}{\partial x_i} = -2(t \arg et_1 - y_1)\frac{\partial y_1}{\partial x_i} - 2(t \arg et_2 - y_2)\frac{\partial y_2}{\partial x_i}$$

$$f_1 = sim(x_1, x_2), f_2 = incl(x_3, x_4)$$

where

$$\frac{\partial y_1}{\partial x_1} = \frac{\partial y_1}{\partial z_2}\frac{\partial z_2}{\partial f_1}\frac{\partial f_1}{\partial x_1}$$

$$\frac{\partial y_1}{\partial x_2} = \frac{\partial y_1}{\partial z_2}\frac{\partial z_2}{\partial f_1}\frac{\partial f_1}{\partial x_2} + \frac{\partial y_1}{\partial z_1}\frac{\partial z_1}{\partial x_2}$$

$$\frac{\partial y_1}{\partial x_3} = \frac{\partial y_1}{\partial z_1}\frac{\partial z_1}{\partial x_3} + \frac{\partial y_1}{\partial z_2}\frac{\partial z_2}{\partial f_2}\frac{\partial f_2}{\partial x_3}$$

$$\frac{\partial y_1}{\partial x_4} = \frac{\partial y_1}{\partial z_2}\frac{\partial z_2}{\partial f_2}\frac{\partial f_2}{\partial x_4}$$

$$\frac{\partial y_2}{\partial x_1} = (w_6 s x_4)(1 - w_5) \qquad \frac{\partial y_2}{\partial x_2} = 0 \qquad \frac{\partial y_2}{\partial x_3} = 0$$

$$\frac{\partial y_2}{\partial x_4} = (w_5 s x_1)(1 - w_6)$$

$$\frac{\partial y_1}{\partial z_1} = z_2 \qquad \frac{\partial y_1}{\partial z_2} = z_1 \qquad \frac{\partial z_1}{\partial x_2} = (x_3 s w_2)(1 - w_1)$$

$$\frac{\partial z_1}{\partial x_3} = (x_2 s w_1)(1 - w_2)$$

$$\frac{\partial z_2}{\partial f_1} = w_3(1 - f_2 w_4) \qquad \frac{\partial z_2}{\partial f_2} = w_4(1 - f_1 w_3)$$

$$\frac{\partial f_1}{\partial x_1} = \frac{\partial}{\partial x_1}[(x_1 \to x_2)(x_2 \to x_1)] = \frac{\partial}{\partial x_1}\begin{cases} x_1 \to x_2 & if x_1 > x_2 \\ 1 & if x_1 = x_2 \\ x_2 \to x_1 & if x_2 > x_1 \end{cases} = \begin{cases} -\frac{x_2}{x_1^2} & if x_1 > x_2 \\ 0 & if x_1 = x_2 \\ \frac{1}{x_2} & if x_2 > x_1 \end{cases}$$

$$\frac{\partial f_1}{\partial x_2} = \begin{cases} -\frac{1}{x_1} & if x_1 > x_2 \\ 0 & if x_1 = x_2 \\ \frac{x_1}{x_2^2} & if x_2 > x_1 \end{cases}$$

$$\frac{\partial f_2}{\partial x_3} = \frac{\partial}{\partial x_3}\begin{cases} 1 & if x_3 \le x_4 \\ \frac{x_4}{x_3} & if x_3 > x_4 \end{cases} = \begin{cases} 0 & if x_3 \le x_4 \\ -\frac{x_4}{x_3^2} & if x_3 > x_4 \end{cases}$$

$$\frac{\partial f_2}{\partial x_4} = \frac{\partial}{\partial x_4}\begin{cases} 1 & if x_3 \le x_4 \\ \frac{x_4}{x_3} & if x_3 > x_4 \end{cases} = \begin{cases} 0 & if x_3 \le x_4 \\ \frac{1}{x_3} & if x_3 > x_4 \end{cases}$$

Noticeably, the learning is fast and usually does not exceed a few hundred epochs. The results of the learning process (construction of the testing oracles) are

- $\boldsymbol{x}$ = [0.002760 0.000000 1.000000 0.999999] for [1 0] with the value of the performance index equal to 0.211608. The resulting output of the network is given as [0.540 0.003] (note that the second output is fully suppressed while the error comes from the departure observed for the first coordinate), and
- $\boldsymbol{x}$ = [0.999081 1.000000 0.308634 1.000000] for [0 1] (in this case the performance index achieved the value equal to 0.041615 and the outputs of the network were equal to [0.203996 0.999030]). In both cases, the starting point of the learning is a random configuration of the values of $\boldsymbol{x}$.

In a nutshell, we note that the testing oracles can be quantified as the following associations (considering the most dominant causes): $\{x_3, x_4\}$ and [1 0] and $\{x_1, x_2, x_4\}$ and [0 1].

The example above leads us to a more formal description of the development of testing oracles and provides us with their interesting numeric characterization. The performance index describes our ability to form a testing oracle for a certain effect. The lower the value of Q, the higher our confidence about the corresponding oracle. Ideally, we would expect Q equal to zero that translates into an ideal test case (oracle). In this case we state that the effect is testable as we can find the corresponding cause vector.

Let us now experiment with several cases considering several collections of the values of the connections of the network; we are interested in quantifying how these changes impact the testing oracles. The results are summarized in Table 2.

Overall, the collections of the test oracles for the selected collections of the connections can be described as follows

- Case 1: no test oracle associated with [1 0] (note that the value of the second output of the network exceeds the one for the first output); test oracle for [0 1] results in $\{x_4\}$ as the most dominant cause.
- Case 2: test oracle for [1 0] consists of $\{x_3, x_4\}$ while $\{x_4\}$ forms the test oracle for [0 1]

- Case 3: [1 0] does not produce any test oracle (the system is not testable). For [0 1] we obtain $\{x_4\}$.

Table 2. Combinations of the connections of the system and the corresponding testing oracles and the associated performance index Q characterizing their quality (learning rate is set to 0.05, number of learning epochs is equal to 200)

w	target [1 0]		target [0 1]	
	Q	x and y~ ; comments	Q	x and y~ comments
$w_1 = 0.0$; $w_2 = 0.1$; $w_3 = 1.0$; $w_4 = 0.4$; $w_5 = 0.2$; $w_6 = 0.4$	0.804	[0.155 0.175 1.000 0.993] [0.163 0.324] testing case not generated; significant deviation from the testing oracle	0.387	[0.225 0.175 0.308 1.00] [0.057637 0.380141]
$w_1 = 0.6$; $w_2 = 0.0$; $w_3 = 1.0$; $w_4 = 0.4$; $w_5 = 0.2$; $w_6 = 0.4$	0.242	[0.158 0.175 1.000 1.000] [0.630 0.325]	0.417	[0.224 0.175 0.308 1.000] [0.180 0.379]
$w_1 = 0.2$; $w_2 = 0.4$; $w_3 = 1.0$; $w_4 = 0.9$; $w_5 = 0.1$; $w_6 = 0.0$	0.497	[0.154 0.175 1.000 0.993] [0.336 0.239] testing case not generated; significant deviation from the testing oracle	0.508	[0.237 0.175 0.308 1.000] [0.194 0.313]

6. Conclusions

We have introduced a new class of logic-oriented computing architectures aimed at the realization of cause-effect testing structures for software systems. It was demonstrated that the direct and inverse problems in such networks correspond to the estimation problem (that is specification refinement) and generation of test oracles. The focus in this study was on gradient-based learning. One possible enhancement being worth considering along the line of the optimization processes would be to consider evolutionary optimization, which may help develop a comprehensive insight into the way possible test oracles are be formed

(which is an attractive option considering the fact of nonuniqueness of the solution to the inverse problem).

References

[1] J.P. Myers, "The complexity of software testing", *Software Engineering Journal*, 1, 1992, 13-24.

[2] W. Pedrycz and F. Gomide, "An Introduction to Fuzzy Sets"; *Analysis and Design*. MIT Press, MA, 1998.

[3] W. Pedrycz, *Computational Intelligence: An Introduction*. CRC Press, Boca Raton, Fl, 1997.

[4] W. Pedrycz, *Fuzzy Sets Engineering*, CRC Press, Boca Raton, Fl, 1995.

[5] W. Pedrycz and J.F. Peters (eds.), *Computational Intelligence in Software Engineering*, World Scientific, Singapore, 1998.

[6] W. E. Perry, *Effective Methods for Software Testing*, J. Wiley, N. York, 2000.

[7] J. F. Peters and W. Pedrycz, *Software Engineering: An Engineering Approach*, J. Wiley, N. York, 1999.

[8] M. L. Shooman, *Reliability of Computer Systems and Networks*, J. Wiley, N. York, 2002.

[9] A. von Mayrhauser, C. W. Anderson, T. Chen, R. Mraz, C.A. Gideon, "On the promise of neural networks to support software testing", In: W. Pedrycz, J.F. Peters (eds.), *Computational Intelligence in Software Engineering*, World Scientific, Singapore, 1998, pp.3-32.

[10] M. Vanmali, M. Last, A. Kandel, "Using a neural network in the software testing process", *Int. J. of Intelligent Systems*, vol. 17, 1, 2002, 45-62.

[11] E. Weyuker, T. Goradia, A. Singh, "Automatically generating test data from a Boolean specification", *IEEE Trans on Software Engineering*, 1994, 20, 5, 353-363.

CHAPTER 2

BLACK-BOX TESTING WITH INFO-FUZZY NETWORKS

Mark Last

Department of Information Systems Engineering
Ben-Gurion University of the Negev
Beer-Sheva 84105, Israel
E-mail: mlast@bgumail.bgu.ac.il

Menahem Friedman

Department of Physics
Nuclear Research Center – Negev
Beer-Sheva, POB 9001, Israel
E-mail: mlfrid@netvision.net.il

Functionality of software code is mostly tested using the black-box approach, where the actual outputs are compared to the expected ones based on the tester's understanding and knowledge of system requirements. Since the available documentation is often incomplete or outdated, especially in the case of a "legacy" application, and the code may be poorly structured, execution data seems to be the most reliable source of information on the real functionality of an evolving system. In complex software applications, manual observation of system inputs and outputs is hardly helpful. In this paper, we demonstrate the potential use of Info-Fuzzy Networks (IFN) for automated induction of functional requirements from execution data. The induced models of tested software can be utilized for recovering missing and incomplete specifications, designing a minimal set of regression tests, and evaluating the correctness of software outputs when testing new, potentially flawed releases of the system. To evaluate the efficiency of the proposed approach, we have applied it to execution data of a sophisticated expert system for solving partial differential equations. Experimental results

demonstrate the clear capability of the IFN algorithm to discriminate between correct and faulty versions of a tested program. In addition, we show the robustness of the info-fuzzy methodology with respect to complex and partially inconsistent data.

1. Introduction

The ultimate goal of *function testing* is to verify that the program works according to its specification and there are no undiscovered errors left. Software functionality is mostly tested using the *black-box* approach, where the actual outputs are compared to the expected ones based on the tester's understanding and knowledge of system requirements. In a properly documented system, behavior requirements are covered by a series of *use-case scenarios* [36]. Use cases are further modeled by *Activity*, *Sequence*, and *Collaboration Diagrams*. Activity diagrams represent a slightly enhanced version of conventional flow charts, while sequence and collaboration diagrams apply mainly to object-oriented systems. The purpose of these requirement models is to completely specify the system actions and outputs in response to inputs obtained from the environment.

If the functional requirements are current, clear, and complete, they can be used as a basis for designing black-box *test cases*. Each test case is a combination of particular input values, events, messages, etc. which should result in pre-specified actions and outputs of the tested system. When requirements can be re-stated as logical relationships between inputs and outputs, test cases can be generated automatically by such techniques as cause-effect graphs (see [36]) and decision tables [2]. T-VEC, an integrated specification and verification method for automatic generation of test vectors from functional relationships, is described in [3]. Since testing resources are always limited and modern software systems are extremely complex, it is impossible to execute a tested program with any feasible combination of inputs. Thus, *effective selection* of test cases and *accurate verification* of their outputs are crucial for improving the *quality* of a software system with *less* cost.

Requirements are rarely static, which means that any software system, whether it is an open source code, an off-the-shelf package or a custom-built application, has to undergo continual changes throughout its life-cycle. Frequent changes make the original functional requirements, even if once complete and accurate, hardly relevant to the new versions of software [10]. To ensure effective design of new test cases, one has to recover (reverse engineer) the real functionality of an evolving system from its execution data.

For complex software applications, manual observation of system interaction with the environment may be a tedious task. In [22], we have presented the idea that functional requirements can be automatically induced from execution data using the IFN (Info-Fuzzy Network) methodology of data mining, which is described in [23] and [27]. In [22] the proposed concept of IFN-based testing has been applied to a small business application. The current study evaluates the effectiveness of the IFN methodology on a sophisticated expert-system application having multiple inputs and outputs of complex nature. In this chapter, we compare between two alternative approaches: inducing a separate IFN model for each system output vs. inducing a single model representing all outputs.

The rest of the chapter is organized as follows. Section 2 provides a brief overview of single-target info-fuzzy networks and describes a new algorithm for inducing multi-target networks. Section 3 presents the info-fuzzy method of input-output analysis and test case verification with a special emphasis on systems having multiple outputs. Section 4 describes the Unstructured Mesh Finite Element Solver (UMFES) program, which is used as our case study. The results of our testing experiments with UMFES are described in Section 5. Finally, Section 6 summarizes the chapter with initial conclusions and directions for future research.

2. Info-Fuzzy Networks

2.1 Single-Target vs. Multi-Target Networks

A comprehensive description of the info-fuzzy network (IFN) methodology for data-driven induction of predictive models is provided in [27]. An info-fuzzy network represents the functional requirements by an "oblivious" tree-like structure, where each input attribute is associated with a single layer (level) and the leaf (terminal) nodes correspond to combinations of input values. Each node in the target (output) layer may be related to a specific value (e.g., *Credit Limit = 10,000*), a continuous range of values (e.g., *Credit Limit = [9500, 11600]*), or an action performed by the program (e.g., approving a credit application).

In a *single-target info-fuzzy network* (IFN), such as the one shown in Figure 1, the target nodes represent distinct values of a single target attribute or a set of *disjoint* actions (such as *credit approval* vs. *credit decline*). This means that we need to re-construct the entire network for each output (target) variable of a tested system. The target layer has no equivalent in decision trees (see [33]), but a similar concept of *category nodes* is used in *decision graphs* [19]. There is no limit as to the maximum number of output nodes in an info-fuzzy network. The single-layer network in Figure 1 has three output nodes denoted by numbers 1, 2, and 3.

A hidden layer No. l consists of nodes representing conjunctions of values of the first l input attributes, which is similar to the definition of an internal node in a standard decision tree. For continuous inputs, the values represent intervals identified automatically by the network construction algorithm. In Figure 1, we have two hidden layers (No. 1 and No. 2). Like in Oblivious Read-Once Decision Graphs [19], all nodes of a hidden IFN layer are labeled by the same input attribute and each input attribute is tested only once along a given network path. However, IFN extends the "read-once" restriction of [19] to continuous input attributes by allowing multi-way splits of a continuous domain at the same network level.

The final (terminal) nodes of the network represent non-redundant conjunctions of input values that produce distinct outputs. The five terminal nodes of the network in Figure 1 are (1,1), (1,2), (2), (3,1), and (3,2). If the network is induced from execution data of a software system, each terminal node represents a non-redundant test case, while interconnections between terminal and target nodes stand for possible outputs of the corresponding test cases. For example, the connection (1,1) → 1 in Figure 1 means that we expect the output value of 2 for a test case where both input variables are equal to 1. The connectionist nature of IFN resembles the structure of a multi-layer neural network. Consequently, we characterize our model as a *network* and not as a *tree*.

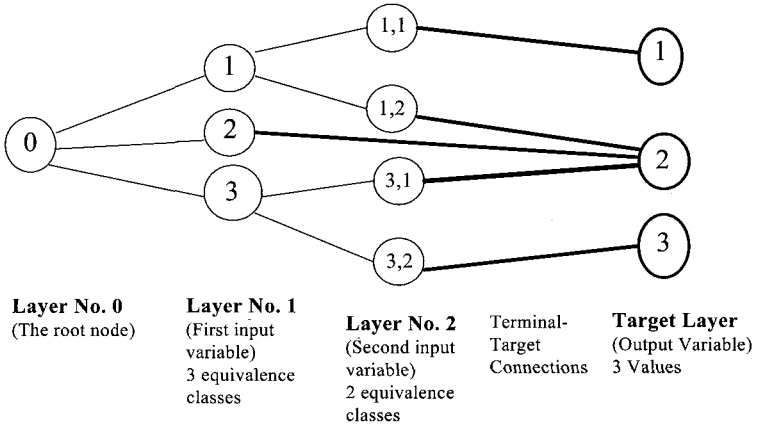


Figure 1. Single-Target Info-Fuzzy Network - An Example

The idea of restricting the order of input attributes in graph-based representations of Boolean functions, called "function graphs", has proven to be very useful for automatic verification of logic design, especially in hardware. Bryant [5] has shown that each Boolean function has a unique (up to isomorphism) reduced function graph representation, while any other function graph denoting the same function contains more vertices. Moreover, all symmetric functions can be represented by function graphs where the number of nodes grows at most as the square of the number of input attributes. As indicated by Kohavi in [18], these

properties of Boolean function graphs can be easily generalized to oblivious read-once decision graphs representing *k*-categorization functions. Consequently, the "read-once" structure of info-fuzzy networks makes them a natural technique for modelling complex software systems.

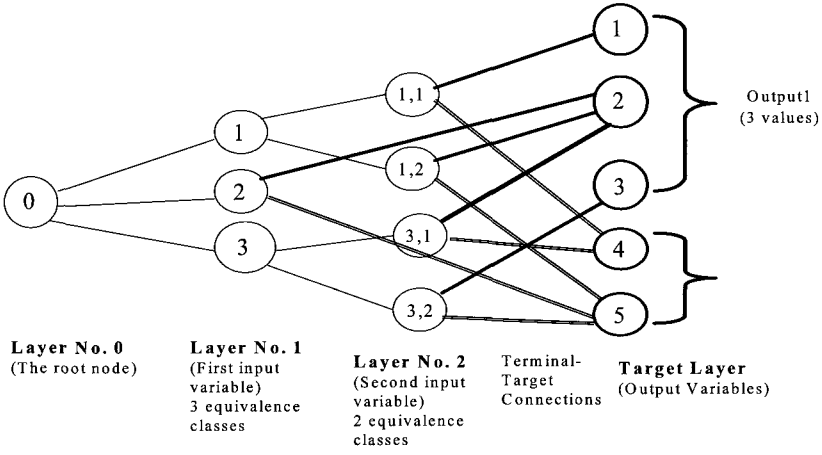


Figure 2. Multi-Target Info-Fuzzy Network - An Example

In a *multi-target info-fuzzy network* (M-IFN), the nodes of the target layer represent values of several output attributes. Figure 2 shows an example of a *multi-target* info-fuzzy network, which has five nodes in its target layer: nodes 1-3 corresponding to the three values of the first output variable (*Output1*) and nodes 4-5 corresponding to the two values of the second output variable (*Output2*). In the network of Figure 2, each terminal node has at least two outgoing connections leading to the expected values of the two output variables. Thus, the terminal node (1, 1) is connected to the first value of *Output1* (represented by node 1) and to the first value of *Output2* (represented by node 4). The internal "read-once" structure of the network is identical for both output variables. In other words, every hidden node is shared among *all* outputs. Thus, M-IFN is an extreme case of a Shared Binary Decision Diagram (SBDD), where some nodes may be shared among several output functions (see [5] and [28]).

The algorithms for inducing single-target networks have been thoroughly described in previous works (see [23] and [27]). A novel algorithm for constructing a multi-target network is presented in the next sub-section.

2.2 Inducing Multi-Target Networks from Data

The main assumption of the multi-target info-fuzzy network (M-IFN) is that a single network represents the required behavior of all output (target) variables. On the contrary, a single-target IFN (see [27]) is completely re-constructed for every target attribute. The M-IFN induction procedure starts with defining the target layer (a node for each target interval or class of every target attribute) and the "root" node representing an empty set of input attributes. The input attributes are selected incrementally, out of a set of "candidate inputs", to maximize a decrease in the total conditional entropy of *all* target attributes. In information theory (see [6]), conditional entropy measures the degree of uncertainty of a given variable, such as a software output, given the values of other variables, which may represent software inputs. If we have a complete and accurate model of system functionality, we know *exactly*, which output values will be produced by every possible combination of system inputs. In this ideal case, the entropy of all outputs will be zero. However, in a real-world situation we may be unable to induce such a perfect model from a given set of execution data. Moreover, as shown by our experiments in [20] and in this study, an imperfect model approximating the actual software system may be quite effective for regression testing purposes.

In the induction algorithm of a single-target IFN (see [27]), we have minimized the conditional entropy of a single target attribute given a set of n input attributes. According to a chain rule proven in [6], the joint entropy of two random variables is equal to the unconditional entropy of the first one plus the conditional entropy of the second one given the first one. This rule can be easily extended to the joint conditional entropy of m output variables given n input variables as follows:

$$H(Y_1, \ldots, Y_m / X_1, \ldots, X_n) = \sum H(Y_i / Y_{i-1}, \ldots Y_1, X_1, \ldots, X_n) \quad (1)$$

However, the exact calculation of Eq. (1) at each step of the network construction algorithm would be very expensive in terms of calculation time and the required number of training examples. To choose the best input attribute associated with the next network layer, we will use the following approximation of (1):

$$\hat{H}\ (Y_1, \ldots, Y_m / X_1, \ldots, X_n) = \sum H\ (Y_i / X_1, \ldots, X_n) \qquad (2)$$

In Eq. (2), we have decomposed the calculation of *joint conditional entropy* of m variables into a sum of m *simple conditional entropies*. Though in a general case, the results of Eqs. (2) may be radically different from (1) leading to a non-optimal choice of the best input attributes, we can prove that in at least two specific situations, which are not uncommon in software systems, the approximate formula (2) will produce *the optimal* solution. These two situations are: 1) all outputs are conditionally independent of each other given the input attributes and 2) there is a functional dependency between all outputs. The formal proof for both cases is provided below.

Lemma 1. If all target attributes are conditionally independent of each other given the input attributes, then Eqs. (1) and (2) are identical.

Proof. According to [6], if A and B are conditionally independent given C, then $H\ (B/A,\ C) = H\ (B/C)$. Consequently, if all output variables Y_i are conditionally independent of each other given the input variables $X_1, \ldots, X_n$ then:

$$\forall i\text{: } H\ (Y_i / Y_{i-1}, \ldots\ Y_1, X_1, \ldots, X_n) = H\ (Y_i / X_1, \ldots, X_n)$$

Which implies that:

$$H\ (Y_1, \ldots, Y_m / X_1, \ldots, X_n) = \sum H\ (Y_i / Y_{i-1}, \ldots\ Y_1, X_1, \ldots, X_n) = \sum H\ (Y_i / X_1, \ldots, X_n)$$

Proof completed.

Lemma 2. If every target attribute is a function of any other target attribute, then Eq. (1) becomes:

$$H\ (Y_1, \ldots, Y_m / X_1, \ldots, X_n) = H\ (Y_1 / X_1, \ldots, X_n)$$

Proof. According to [6], if there is a functional dependency between the variables A and B ($B = f\ (A)$), then $H\ (B / A) = 0$, which means that the value of B is certain given any value of A. This implies that

$\forall i > 1$: $H(Y_i / Y_{i-1}, \ldots Y_1, X_1, \ldots, X_n) = 0$.

Consequently, Eq. (1) becomes

$H(Y_1, \ldots, Y_m / X_1, \ldots, X_n) = H(Y_1 / X_1, \ldots, X_n)$

Proof completed.

Theorem 1. In cases covered by Lemmas 1 and 2, the input attributes selected by minimizing Eq. (2) will minimize the joint conditional entropy of all output variables as calculated by Eq. (1).

Proof. When the conditions of Lemma 1 are satisfied, the results of both equations will be equal leading automatically to an identical choice of the best input attribute. In case of Lemma 2 conditions, the functional dependency of target attributes implies that

$H(Y_1 / X_1, \ldots, X_n) = H(Y_2 / X_1, \ldots, X_n) = \ldots = H(Y_m / X_1, \ldots, X_n)$

Based on Lemma 2 and the above result, Eq. (2) becomes

$\hat{H}(Y_1, \ldots, Y_m / X_1, \ldots, X_n) = m \cdot H(Y_1 / X_1, \ldots, X_n)$

In other words, the result of Eq. (2) is a multiplier of Eq. (1). This implies that a set of input attributes providing a maximum decrease in the approximated conditional entropy calculated by Eq. (1) will also be identified as the best set using Eq. (2). Proof completed.

Unlike CART™ [4], C4.5 [37], and EODG [19], the M-IFN induction algorithm is based on the pre-pruning approach: when no input attribute causes a statistically significant decrease in the conditional entropy, the network construction is stopped. In this paper, we focus on the selection of continuous input attributes, which present in our case study. The treatment of discrete input attributes by the M-IFN algorithm is a subject of ongoing research.

The algorithm performs discretization of continuous input attributes "on-the-fly" by using an approach, which is based on the information-theoretic heuristic of Fayyad and Irani [11]: recursively finding a binary partition of an input attribute that minimizes the total conditional entropy of all target attributes. However, the stopping criterion we are using is different from [11]. Rather than searching for a minimum description length (minimum number of bits for encoding the training data), we make use of a standard statistical likelihood-ratio test [38] with respect to the conditional entropy of every target attribute. If the test fails to reject

the null hypothesis for the *i*-th target attribute, the corresponding term $H(Y_i / X_1, ..., X_n)$ in Eq. (2) is considered as equal to zero. The search for the best partition of a continuous attribute is dynamic: it is performed each time a candidate input attribute is considered for inclusion in the network. After discretization, each hidden node in the network is associated with an interval of a continuous input attribute.

For each target attribute A_i, we evaluate the statistical significance of splitting the interval S by the threshold T at a node z using the likelihood-ratio statistic (based on [38]):

$$G^2(T, A_i / S, z) = 2 \sum_{j=0}^{M_i - 1} \sum_{y=1}^{2} N_{ij}(S_y, z) \bullet \ln \frac{N_{ij}(S_y, z)}{P(C_{ij} / S, z) \bullet E(S_y, z)} \qquad (3)$$

Where

M_i is the number of target nodes associated with the attribute A_i. Each target node represents a value j of the attribute A_i;

$N_{ij}(S_y, z)$ is the number of occurrences of the target value j in sub-interval S_y at a node z;

$E(S_y, z)$ is the number of records in sub-interval S_y at a node z;

$P(C_{ij} / S, z)$ is an estimated conditional (a posteriori) probability of the target value j given the interval S and the node z; and

$P(C_{ij} / S, z) \bullet E(S_y, z)$ - an estimated number of occurrences of the target value j in sub-interval S_y at a node z under the assumption of the null hypothesis that the conditional probabilities of the target attribute values are identically distributed in each sub-interval.

The Likelihood-Ratio Test is a general-purpose method for testing the null hypothesis H_0 that two random variables are statistically independent. If H_0 holds, then the likelihood-ratio test statistic $G^2(T, A_i / / S, z)$ is distributed as chi-square with $NT(S, z) - 1$ degrees of freedom, where $NT(S, z)$ is the number of values taken by the target attribute in the interval S at node z. The default significance level (*p-value*) used by the M-IFN algorithm, is 0.1%.

If the result of Eq. (3) proves statistically significant, we mark the node z as "split" and calculate the estimated decrease in the conditional entropy of the target attribute A_i (termed as *conditional mutual information*) by the following formula (based on [6]):

$$MI(T;A_i/S,z)=$$
$$\sum_{j=0}^{M_i-1}\sum_{y=1}^{2}P(S_y;C_{ij};z)\cdot\log\frac{P(S_y;C_{ij}/S,z)}{P(S_y/S,z)\cdot P(C_{ij}/S,z)} \quad (4)$$

Where

$P(S_y; C_{ij}; z)$ is an estimated joint probability of a value j of the target attribute A_i, a sub-interval S_y, and the node z;

$P(S_y; C_{ij} / S, z)$ is an estimated joint probability of a value j of the target attribute A_i and a sub-interval S_y given the interval S and the node z;

$P(S_y / S, z)$ is an estimated conditional (a posteriori) probability of a sub-interval S_y given the interval S and the node z; and

$P(C_{ij} / S, z)$ is an estimated conditional (a posteriori) probability of a value j of the target attribute A_i given the interval S and the node z.

The range of an input attribute is split at a threshold T, which maximizes the total conditional mutual information over all nodes and all target attributes. The binary discretization process is repeated recursively as long as the resulting decrease in the total conditional entropy is greater than zero. A new input attribute is selected to maximize the total significant decrease in the conditional entropy as a result of splitting the nodes of the last layer. The nodes of a new hidden layer are defined for a Cartesian product of split nodes of the previous hidden layer and discretized intervals of the new input variable. If there is no candidate input variable significantly decreasing the total conditional entropy of the output variables, the network construction stops. In Figure 2, the first hidden layer has three nodes related to three intervals of the first input variable, but only nodes 1 and 3 are split, since the total conditional mutual information as a result of splitting node 2 proves to be statistically insignificant. For each split node of the first layer, the algorithm has created two nodes in the second layer, which represent the two intervals of the second input variable. None of the four nodes of the second layer are split, because they do not provide a significant decrease in the conditional entropy of the outputs.

Table 1. Multi-Target Network Construction Algorithm

Step 2.4	End Do
Step 3	Return the set of selected inputs *I*, the associated discretization classes, and the network structure
Input:	The set of n training examples; the set C of candidate inputs; the set O of target (output) variables; the minimum significance level sign for splitting a network node (default: sign = 0.1%).
Output:	A set I of selected inputs, discretized intervals for every input, and an info-fuzzy network. Each selected input has a corresponding hidden layer in the network.
Step 1	Initialize the info-fuzzy network (single root node representing all examples, no hidden layers, and a target layer for the values of the output variables). Initialize the set I of selected inputs as an empty set: I = ∅.
Step 2	While the number of layers |I| < |C| (number of candidate inputs) do
Step 2.1	For each candidate input Ai' /Ai' ∈ C; Ai' ∉ I do
Step 2.1.1	For each distinct value T included in the range of Ai' (except for the last distinct value) Do: Initialize the total conditional mutual information given the threshold T to zero For each node z of the final hidden layer Do: For each target attribute Ai Do: Calculate the likelihood-ratio test for the partition of the interval S at the threshold T vs. the target attribute Ai given the node z. If the likelihood-ratio statistic is significant, mark the node as "split" by the threshold T and increment the total conditional mutual information given the threshold T End Do End Do End Do
Step 2.1.2	Find the threshold Tmax maximizing the total conditional mutual information over all nodes and all target attributes

Step 2.1.3	If the maximum conditional mutual information is greater than zero, then Do:
	For each node z of the final hidden layer Do:
	If the node z is split by the threshold Tmax, mark the node as split by the candidate input attribute Ai'
	Partition each sub-interval of S (go to Step 2.1.1)
	End Do
	End Do
	Else
	Define a new interval of Ai'

Step 2.1.4	End Do

Table 1. Multi-Target Network Construction Algorithm (cont.)

Step 2.2	Find the candidate input Ai'* maximizing cond_MIi'

Step 2.3	If cond_MIi'* = 0, then
	End Do.
	Else
	Expand the network by a new hidden layer associated with the input Ai',
	and add Ai' to the set I of selected inputs $I = I \cap Ai'$.

In Table 1, we show the main steps for constructing a multi-target info-fuzzy network from a set of continuous input attributes.

The M-IFN induction procedure shown in Table 1 is, like its single-target counterpart, a greedy algorithm, which is not guaranteed to find the optimal ordering of input attributes. Moreover, different orderings and network structures may be optimal for different output variables. In [23], we have shown that single-target models induced by the IFN algorithm tend to be nearly as accurate as the best known data mining models for the corresponding target attributes. A reasonably high predictive accuracy of IFN models is important if we intend to use them as "automated oracles" in regression testing, but expecting them to become "perfect predictors" of all outputs in complex software systems is certainly unrealistic. On the other hand, as shown in [22], the inherent compactness of these models can help us to recover the most dominant requirements from execution data and, consequently, to build a compact set of test cases. In this paper, we intend to obtain even more compact models of multi-output software by inducing a common network for all outputs.

3. Black-Box Testing with Single-Target and Multi-Target Info-Fuzzy Networks

The architecture of the IFN-based environment for automated black-box testing is shown in Figure 3. Random Tests Generator (RTG) obtains the list of system inputs, their types (discrete, continuous, etc.), and ranges from System Specification. No information about the functional requirements is needed, since IFN induction algorithms automatically

reveal input-output relationships from randomly generated test cases. In our previous experiments (see [20]), we have found that a relatively small number of 1,000 training cases is sufficient to obtain a reasonably accurate model of the tested system. Systematic, non-random approaches to training set generation may also be considered.

IFN algorithms are trained on execution data, which includes the inputs provided by the RTG module and the corresponding outputs obtained from the legacy version of the tested system by means of the Test Bed module. Our methodology builds upon a common assumption in regression testing that the legacy version is stable, i.e. it has been in use for a sufficiently long period of time to eliminate nearly all errors. As indicated above, we can induce a separate (single-target) IFN model for each output variable or a common (multi-target) model for all output variables. The following information can be derived from an induced IFN model:

(i) A list of input attributes relevant to each output (in the single-target case) or to at least one output (in the multi-target case). Since each input attribute is associated with exactly one network layer, this list can be easily derived from the structure of each induced network.

(ii) Logical (if... then...) rules expressing the functional requirements as logical relationships between the selected input attributes and the system outputs. The set of rules connecting a given terminal node to a set of target nodes represents the distribution of output values expected at that node (see [27]). These rules can be used to determine the predicted (most probable) value of each output in a test case corresponding to each terminal node. Thus, an induced network can be used as an "automated oracle" when testing a later version of the system.

(iii) Discretization intervals of each continuous input attribute included in the network. In testing terms, each interval represents an "equivalence class", since for all values of a given interval the output values follow the same distribution. When building a set of single-target models, the discretization intervals may vary between target attributes.

(iv) A set of non-redundant test cases to be stored in the regression test library. As indicated above, each terminal node in the network can be converted into at least one test case representing a non-redundant conjunction of input values.

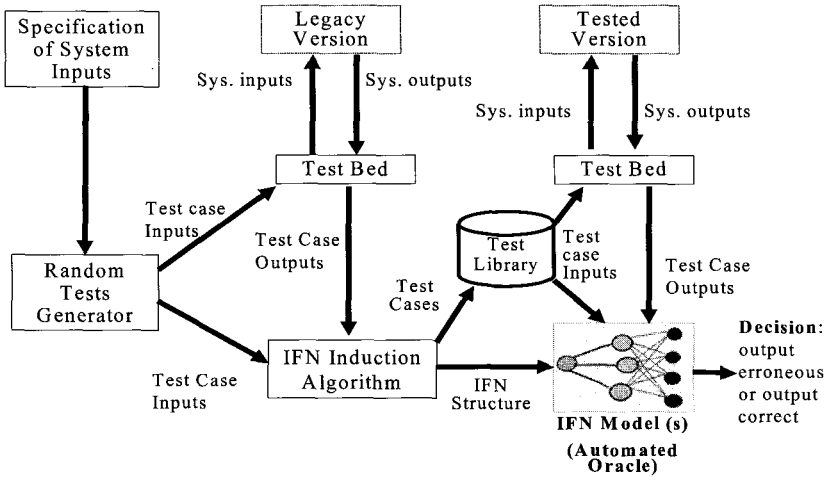


Figure 3. M-IFN Based Black-Box Testing

The role of each module in the testing environment is summarized below:

Specification of System Inputs (SSI). Basic data on each input variable includes variable name, type (discrete, continuous, nominal, etc.), and a list or a range of possible values. Such information is generally available from requirements management and test management tools (e.g., Rational RequisitePro® or TestDirector®).

Random Tests Generator (RTG). This module generates random combinations of values in the range of each input variable. Variable ranges are obtained from the SSI module (see above). The number of test cases to generate is determined by the user. The generated test cases are used by the Test Bed and the IFN modules.

Test Bed (TB). This module, sometimes called "test harness", feeds test cases generated by the RTG module to the software system, which may be a legacy version (when training an IFN algorithm) or a new,

tested version (when using IFN models as automated oracles). The module obtains the system outputs for each test case and sends them to the Induction Algorithm or an induced IFN model.

Info-Fuzzy Network Induction Algorithm (IFN). This module currently includes two algorithms: the one for building a single-target network of a given output (see [23]) and the other one for building a multi-target network for all outputs (see sub-section 2.2 above). The input to each IFN algorithm includes test cases randomly generated by the RTG module and outputs produced by the legacy version for each test case. IFN also uses the descriptions of variables stored by the SSI module. As explained above, the single-target IFN algorithm is run repeatedly to build a network corresponding to each output. The multi-target IFN algorithm is run only once for all outputs. Actual test cases are generated from the automatically detected equivalence classes by using an existing testing policy (e.g., one test for each side of every equivalence class).

In [22] we have applied the single-target IFN algorithm to execution data of a small business application (Credit Approval), where the algorithm has chosen 165 representative test cases from a total of 11 million combinatorial tests. The Credit Approval application had 8 mostly discrete inputs and two outputs (one of them binary). It was based on well-defined logic rules implemented in less than 300 lines of code. In [20], we have applied the single-target algorithm to the basic version of an expert system software for solving partial differential equations. In this chapter, we compare the performance of the single-target and the multi-target algorithms in testing more complex versions of the same system.

4. Case Study: A Finite Element Program for Solving Differential Equations

4.1 The Finite Element Method

The finite element method for solving differential equations was introduced in the late 1960's as a problem solver in mechanical

engineering [31, 43] and quickly became a powerful tool in mathematics, physics, and engineering sciences [12, 14, 30]. The method includes two stages. The first is finding a "functional" which is usually an integral that includes the input of the problem and an unknown function, and is minimized by the solution of the differential equation. The existence of such a functional is a necessary condition for implementing the finite element method. The second stage is partitioning the domain, over which the equation is solved, into local "elements", usually triangles. This process is called "triangulation" and it provides the "finite elements". Over each element the solution is approximated by a low degree polynomial (usually no more than fourth-order).

The coefficients of each polynomial are determined at the end of the minimization process. The union of the polynomials attached to all the finite elements provides an approximate solution to the problem. As in finite differences, a finer "finite element mesh" will provide a more accurate approximation of the solution.

4.2 The Problem: Laplace Equation

In this study, we consider solving the Laplace equation

$$\nabla^2 \phi = \phi_{xx} + \phi_{yy} = 0 \tag{5}$$

over the unit square

$$D = \{(x, y) \mid 0 \le x \le 1, 0 \le y \le 1\} \tag{6}$$

where the solution equals some given $f(x, y)$ (for example, electric potential) on the three of the square's four sides and a homogeneous Neumann boundary condition $\partial\phi / \partial n = 0$ (no electric flux in the same example) is requested along the fourth side. This problem, i.e. finding $\phi(x, y)$ inside the square, has a unique solution. To find it we first define the functional

$$F = \iint_D (\phi_x^2 + \phi_y^2) dx dy \tag{7}$$

where $\phi(x, y)$ is an arbitrary function which equals $f(x, y)$ on the square's three sides. This functional is guaranteed to attain its minimum

at $\phi = \phi_0$ where ϕ_0 is the exact solution. The fact that the minimum solution, which obviously equals $f(x, y)$ along three sides, also satisfies the boundary condition $\partial\phi/\partial n = 0$ over the fourth side emphasizes that a homogeneous Neumann boundary condition is a *natural boundary condition* of the particular functional given by Eq. (7), i.e. a boundary condition that exists without being explicitly imposed.

We now triangulate the square as shown in Figure 4. The triangulation is carried in a manner that guarantees that none of the angles will be 'too close' to 0 or 180 degrees. This increases the approximate solution accuracy.

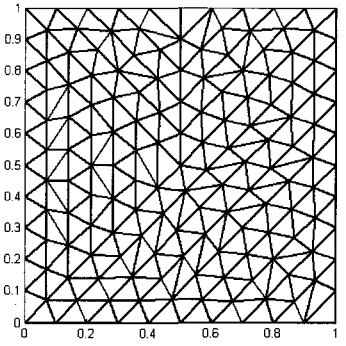


Figure 4. Triangulation of the Domain

A low degree polynomial (say, linear) is chosen to approximate the solution over each triangle. Their unknown coefficients are determined by minimizing the functional (Eq. (7)) subject to the boundary conditions

$$\phi = f \tag{8}$$

given over three of the square's four sides.

The given problem is somewhat an extension of a previous case [20], where the solution was given along the complete boundary (*Dirichlet Boundary Conditions*). Practical problems in engineering usually consist of both types of boundary conditions.

4.3 The Software

We applied an Unstructured Mesh Finite Element Solver (UMFES) which is a general finite element code for solving 2D elliptic partial differential equations (e.g., Laplace's equation) over an arbitrary bounded domain. UMFES was coded in FORTRAN 77 consisting of about 3,000 lines. The first part of the code can triangulate any given bounded two-dimensional domain, while the second part applies the finite element method to solve a given 2-D partial differential equation. The input of each problem includes the boundary conditions (B.C.), i.e. information about the solution along the domain's boundary. The code can treat three types of boundary conditions. The first one is Dirichlet type, where the solution ϕ is a specified function over some portion of the boundary. The second type is called homogeneous Neumann and states $\partial\phi / \partial n = 0$ over another portion of the boundary. The third is a mixed type B.C., represented by the relation $\partial\phi / \partial n + \sigma\phi = \eta$ where σ and η are functions, given along a third portion of the boundary. The finite element mesh is created by a fuzzy expert system [13].

The mesh points which are the triangles' vertices are numbered such that each two points whose numbers are algebraically close, are also geometrically close, 'as much as possible'. This important feature guarantees that the integration matrix, attained by calculating the functional, which combines the geometric structure of the triangular mesh, the coefficients of the differential equations and the boundary conditions, and whose inverse is calculated at the final stage of the numerical process, is such that all its nonzero elements are located within a narrow strip around the main diagonal, an advantageous property for large matrices.

The rules for triangulation, based on knowledge accumulated over many years of experience, were chosen not only to minimize the process and to increase the accuracy of the approximate solution but also to guarantee that the procedure does not reach a dead end. The density of the finite elements is supplied by the user and should be correlated to some preliminary knowledge of the solution's features such as the sub-

domains where it varies slow or fast. A typical inhomogeneous triangulation is shown in Figure 4a. In our initial experiments, described in the next section, we have used homogeneous density.

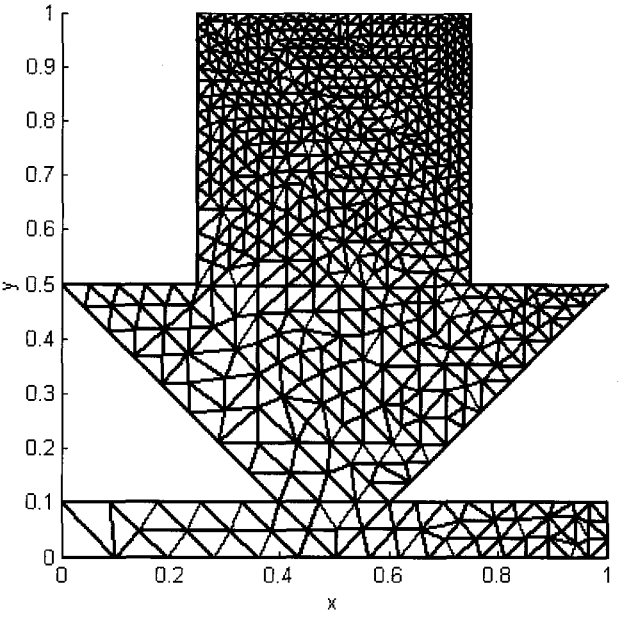


Figure 4a. Inhomogeneous Grid

5. Empirical Results

5.1 Design of Experiments

We have applied UMFES to the problem of sub-section 4.2, where $f(x,y)$ was replaced by constants. Each problem was associated with boundary conditions defined by four constants, which prefixed the solution over the sides of the unit square. The outputs were taken as the solution's values at the fixed points:

$(1/4,1/4),(3/4,1/4),(3/4,3/4),(1/4,3/4),(1/2,1/2)$

We have denoted these outputs (target attributes) by *Out1*... *Out5* respectively. The minimal distance from the boundary affects the predictive accuracy of the numerical solution. Thus, we can expect that the prediction for the last point (*Out5*) will be the least accurate one. We chose a sufficiently fine mesh of elements that guaranteed at least 1% accuracy at each of the output. Thus, a typical problem was represented by a vector of 9 components, 4 inputs and 5 outputs.

We first solved 1,000 problems for training the IFN induction algorithms. Our previous experiments [20] have shown this number of training examples to be sufficient for attaining nearly highest possible accuracy of the induced model. We distinguished between a problem where the solution was known along the complete boundary (*Dirichlet problem*) and a problem where the solution was known along three arbitrary randomly chosen sides with zero normal derivative along the fourth side (*Neumann problem*). The Dirichlet B.C. were taken randomly between -10 and 10. A random number generator was applied to define the problem's type and we ended up with 497 Neumann problems and 503 Dirichlet problems. This training data has been used to build six IFN models: five *single-target models* each representing one software output and one multi-target model representing all five outputs.

The next step was to evaluate the capability of the IFN models to detect an error in a new, possibly faulty version of the program. For this purpose, we have created three sets of 500 validation problems which were different from the 1,000 training problems: (a) mixed Dirichlet / Neumann problems (evenly divided) (b) pure Dirichlet problems, and (c) pure Neumann problems. After solving these problems with the original (stable) version of the software, we have processed them with four "faulty" versions of the original code, where the outputs were mutated *within the boundaries of the solution range* as follows:

(i) error = 0.25*rand [-10, 10], where rand [-10, 10] is a random number in the range of the solution
(ii) error = 0.50*rand [-10, 10]
(iii) error = 0.75*rand [-10, 10]
(iv) rand [-10, 10]

As indicated above, the input for each Dirichlet problem includes four numerals, which represent the solution values over the four sides. In the case of a Neumann problem, we have attached a 'very large' fictitious value of 10^8 to the respective side. This value indicates that the problem is a Neumann problem and merely triggers a certain part of the finite element program that treats the problem as such. It does not present a numerical quantity as is the case for a Dirichlet problem. Consequently, the IFN algorithms can identify this value as a separate interval for all Neumann problems. We have called these problems "Exact Neumann" problems. To present a further challenge to the IFN algorithms, we have created additional 1000 training problems of Neumann and Dirichlet types where the value 10^8 was replaced by a random value between $10^8 - 10^6$ and $10^8 + 10^6$. These problems were called "Randomized Neumann". The performance of the IFN models induced from these problems was validated using the same sets of problems that were created for validating the "Exact Neumann" models.

5.2 Summary of Results

Our first research question is: "What is the difference between the single-target and the multi-target models in terms of predictive accuracy and model complexity?" Since the results were quite consistent across different sets of training and validation problems, we show in Figure 5 the validation RMSE (Root Mean Square Error) of models induced from 1000 mixed (50% "Exact Neumann" / 50% Dirichlet) problems and validated on 500 mixed problems solved by the correct version of the code. It is clear from Figure 5 that the multi-target model of this program is significantly less optimal than the single-target models constructed individually for every output. The increase in the mean prediction error as a result of using a multi-target model varies between 19 % for *Out5* and three times higher for *Out4*. On average, the mean error of the multi-target model is more than two times higher than the error of the single-target models. While the single-target models tend to be more accurate, the main advantage of the multi-target model is its compactness. If our test set would be based on the five single-target

models, the minimal number of test cases would be 388 (one per each terminal node). On the other hand, the multi-target model induced from the same data includes 135 terminal nodes only, which is about one-third of the size of the single-target test set. The effectiveness of fault detection using both types of models has been evaluated in our further experiments.

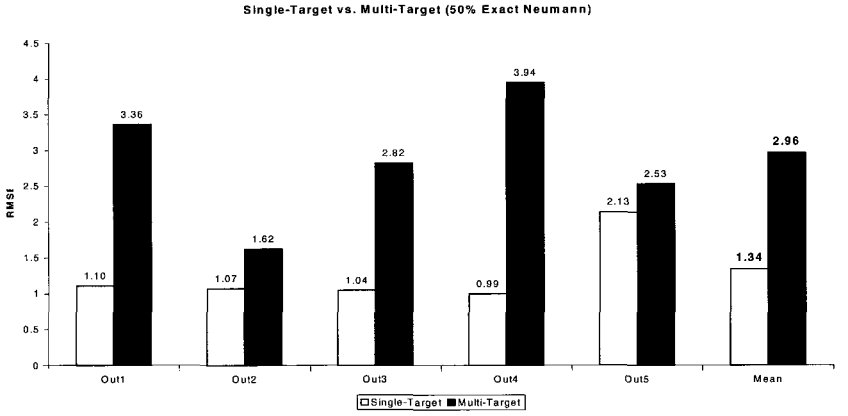


Figure 5. Validation Error of Single-Target vs. Multi-Target Model

In Figures 6 – 8, we compare the mean error of the multi-target model on 500 problems solved by the correct version of the program to its error on four faulty versions of varying severity (see above). We can see that even for the smallest injected error (1/4 of the solution range) there is a significant difference between the RMSE of the outputs generated by the original program and the RMSE of the outputs generated by the faulty program. This includes the "hard to predict" *Out5* attribute. Thus, the relatively low accuracy of the multi-target model does not cripple its capability to discriminate between correct and faulty versions of tested software.

In the third part of our experiments, we have studied predictive accuracy of single-target and multi-target IFN models as a function of the problem types solved by the tested software. Our basic assumption is that a model induced from one type of problems may be used to test a new software version on a different type of problems. We have

considered the following problem types for training and validating our algorithms:

- Pure Dirichlet (100% Dirichlet problems)
- Mixed Exact Neumann (50% Dirichlet / 50% Exact Neumann)
- Mixed Randomized Neumann (50% Dirichlet / 50% Randomized Neumann)
- Pure Exact Neumann (100% Exact Neumann problems)

Out of 16 possible combinations of training and validation problems, we have chosen the following five representative combinations:

- Pure Dirichlet / Pure Dirichlet
- Mixed Exact Neumann / Mixed Exact Neumann
- Mixed Exact Neumann / Pure Exact Neumann
- Mixed Exact Neumann / Dirichlet
- Mixed Randomized Neumann / Mixed Exact Neumann

The mean errors of single-target and multi-target models for each combination are shown in Figures 9 and 10 respectively. The outputs of validation problems were produced with the correct version of the software. The main conclusion from both charts is that the prediction error of IFN models is more sensitive to the problem it is trying to solve (the validation problem) than to the problems that were used to build these models (the training problems). In other words, IFN induction algorithms are quite robust with respect to complex and partially inconsistent training data. Thus in Figure 9 we can see that the curves of Dirichlet / Dirichlet and Mixed Exact Neumann / Dirichlet are much closer to each other than the curves of Mixed Exact Neumann / Mixed Exact Neumann and Mixed Randomized Neumann / Mixed Exact Neumann. The curve of Mixed Exact Neumann / Pure Exact Neumann is located above the other four curves and is clearly the most difficult problem to predict. A similar picture has been obtained for the multi-target models with a notable exception of Dirichlet / Dirichlet, which had a completely different behavior than the other four curves. In our further research, we will be trying to understand this phenomenon.

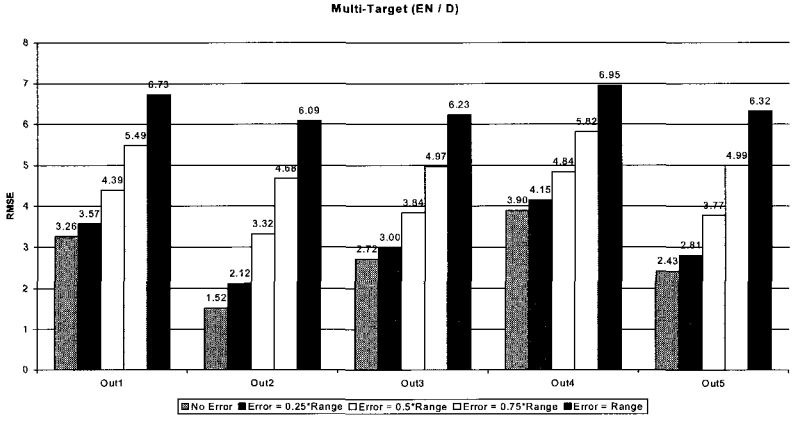


Figure 6. Validation Error of Multi-Target Model over Faulty Versions (100% Dirichlet Problems)

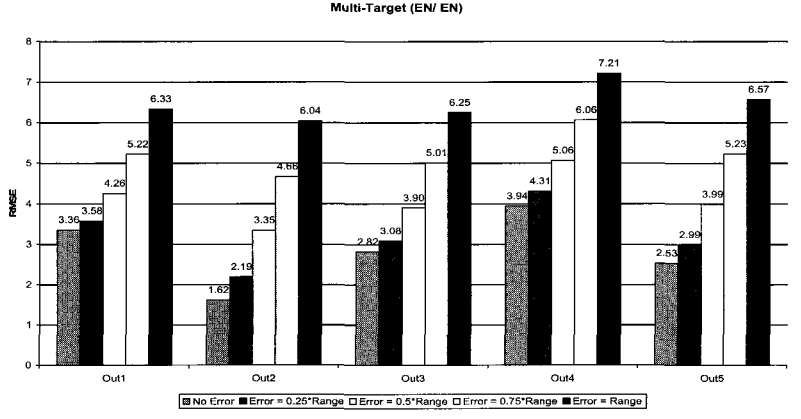


Figure 7. Validation Error of Multi-Target Model over Faulty Versions (50% Exact Neumann / 50% Dirichlet Problems)

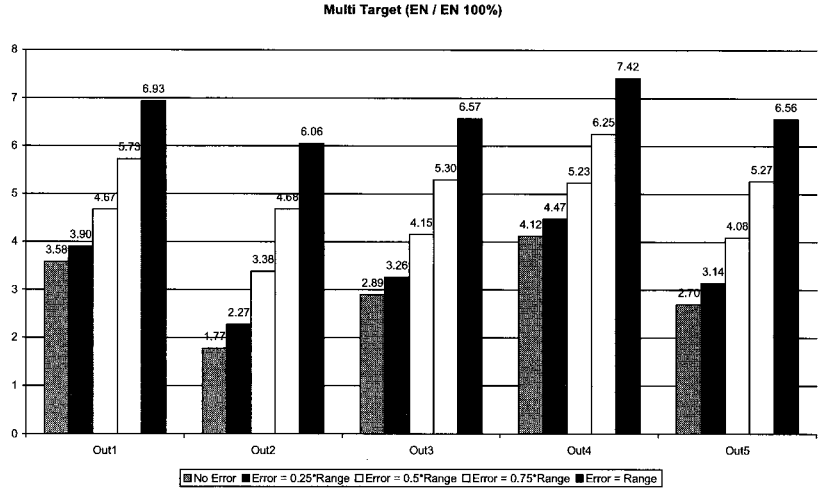


Figure 8. Validation Error of Multi-Target Model over Faulty Versions (100% Exact Neumann Problems 5. Validation Error of Single-Target vs. Multi-Target Model

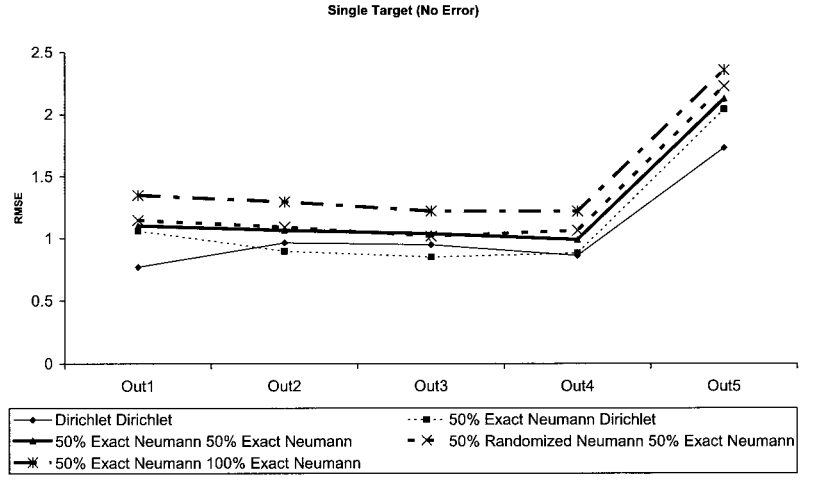


Figure 9. Validation Error of Single-Target Models as a Function of Problem Type

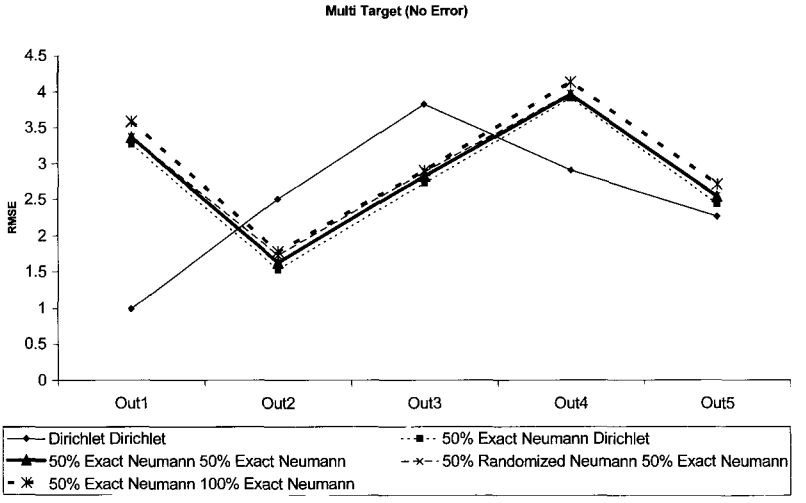


Figure 10. Validation Error of Multi-Target Models as a Function of Problem Type

6. Conclusions

In this chapter, we have presented and evaluated an emerging DM-based methodology for automated black-box testing of evolving software systems. The method has been applied to inputs and outputs of a sophisticated expert system for solving partial differential equations. We have shown that the proposed algorithms can handle the following problems associated with regression testing of data-driven software:

- The IFN algorithms can automatically produce a set of non-redundant test cases covering the most common functional relationships existing in software (including the corresponding equivalence classes). Existing methods and tools for automated software testing do not provide this capability.
- The method is applicable to testing complex software systems, since it does not depend on the analysis of system code like the white-box methods of automated test case selection.

- No significant human effort is required to use the method. Current methods and techniques of test case design assume manual analysis of either the requirements, or the code.
- Missing, outdated, or incomplete requirements are not an obstacle for the proposed method, since it induces the functional relationships automatically from the execution data itself. Existing methods of test case generation assume knowledge of requirements, which may be unavailable or incomplete for many legacy systems.

Issues for further experimentation with Finite Element Solver include introduction of nominal attributes indicating presence or absence of Neumann boundary conditions, fault injection in the code itself rather than in program outputs, and detecting faults in dynamic (time-dependent) problems such as bubble expansion. We also intend to develop new, more optimal algorithms for modeling multi-input, multi-output applications.

Acknowledgement

This work was partially supported by the National Institute for Systems Test and Productivity at University of South Florida under the USA Space and Naval Warfare Systems Command Grant No. N00039-01-1-2248.

References

[1] "Astra QuickTest from Mercury Interactive" http://astratryandbuy.mercuryinteractive.com
[2] Beizer, B. *Software Testing Techniques*. 2nd Edition, Thomson, 1990.
[3] Blackburn, M.R., Busser, R.D., Fontaine, J.S. "Automatic Generation of Test Vectors for SCR-Style Specifications". *Proceedings of the 12th Annual Conference on Computer Assurance* (Gaithersburg, Maryland, June 1997).
[4] Breiman, L., Friedman, J.H., Olshen, R.A., and Stone, P.J. *Classification and Regression Trees*. Wadsworth, 1984.
[5] Bryant, R. E. "Graph-Based Algorithms for Boolean Function Manipulation". *IEEE Transactions on Computers,* C-35-8, 677-691, 1986.
[6] Cover, T. M., and Thomas, J.A. *Elements of Information Theory*. Wiley, 1991.
[7] DeMillo R.A., and Offlut, A.J. "Constraint-Based Automatic Test Data Generation". *IEEE Transactions on Software Engineering*, 17, 9, 900-910, 1991.

[8] Dustin, E., Rashka, J., Paul, J. *Automated Software Testing: Introduction, Management, and Performance*. Addison-Wesley, 1999.

[9] Elbaum, S., Malishevsky, A. G., Rothermel, G. Prioritizing "Test Cases for Regression Testing". *Proc. of ISSTA* '00, 102-112, 2000.

[10] El-Ramly, M., Stroulia, E., Sorenson, P. From "Run-time Behavior to Usage Scenarios: An Interaction-pattern Mining Approach". *Proceedings of KDD-2002* (Edmonton, Canada, July 2002), ACM Press, 315 – 327.

[11] Fayyad, U., and Irani, K. "Multi-Interval Discretization of Continuous-Valued Attributes for Classification Learning". *Proc. Thirteenth Int'l Joint Conference on Artificial Intelligence* (San Mateo, CA, 1993), 1022-1027.

[12] Friedman, M., Rosenfeld, Y., Rabinovitch A., and Thieberger, R. "Finite Element Methods for Solving the Two Dimensional Schrödinger Equation". *J. of Computational Physics*, 26, 2, 1978.

[13] Friedman, M., Schneider, M., Kandel, A. *FIDES – Fuzzy Intelligent Differential Equation Solver*. Avignon, 1987.

[14] Friedman, M., Strauss, M., Amendt, P., London R.A., and Glinsky, M.E. "Two-Dimensional Rayleigh Model For Bubble Evolution in Soft Tissue". *Physics of Fluids*, 14, 5, 2002.

[15] Hamlet, D. "What Can We Learn by Testing a Program?" *Proc. of ISSTA* 98, 50-52, 1998.

[16] Hildebrandt, R., Zeller, A. "Simplifying Failure-Inducing Input". *Proc. of ISSTA* '00, 135-145, 2000.

[17] Kaner, C., Falk, J., Nguyen, H.Q. *Testing Computer Software*. Wiley, 1999.

[18] Kohavi R. "Bottom-Up Induction of Oblivious Read-Once Decision Graphs". Proceedings of the ECML-94, *European Conference on Machine Learning* (Catania, Italy, April 6-8, 1994), 154-169.

[19] Kohavi R., and Li, C-H. "Oblivious Decision Trees, Graphs, and Top-Down Pruning". *Proc. of International Joint Conference on Artificial Intelligence* (IJCAI), 1071-1077, 1995.

[20] Last M., Friedman, M., and Kandel, A. *The Data Mining Approach to Automated Software Testing*, submitted to publication.

[21] Last M., and Kandel, A. "Automated Quality Assurance of Continuous Data". *NATO Advanced Research Workshop on Systematic Organisation of Information in Fuzzy Systems* (Vila Real, Portugal, October 25-27), 2001.

[22] Last M., and Kandel, A. *Automated* "Test Reduction Using an Info-Fuzzy Network". to appear in *Annals of Software Engineering, Special Volume on Computational Intelligence in Software Engineering*, 2003 .

[23] Last M., and Maimon, O. "A Compact and Accurate Model for Classification". to appear in *IEEE Transactions on Knowledge and Data Engineering*, 2003.

[24] Last, M. "Online Classification of Nonstationary Data Streams". *Intelligent Data Analysis*, 6, 2, 129-147, 2002.

[25] Last, M., Kandel, A., Maimon, O. "Information-Theoretic Algorithm for Feature Selection". *Pattern Recognition Letters*, 22 (6-7), 799-811, 2001.

[26] Last, M., Maimon, O., Minkov, E. "Improving Stability of Decision Trees". *International Journal of Pattern Recognition and Artificial Intelligence*, 16, 2, 145-159, 2002.

[27] Maimon O., and Last, M. Knowledge *Discovery and Data Mining – The Info-Fuzzy Network (IFN) Methodology*. Kluwer Academic Publishers, Massive Computing, Boston, December 2000.

[28] Matsuura M., Sasao, T., Butler, J.T., and Iguchi, Y. "Bi-partition of shared binary decision diagrams, Workshop on Synthesis and System Integration of Mixed Technologies" (*SASIMI-2001*), Nara, Japan, Oct. 18-19, 2001, pp.172-177.

[29] Mayrhauser, A. von, Anderson, C.W., Chen, T., Mraz, R., Gideon, C.A. "On the Promise of Neural Networks to Support Software Testing". In W. Pedrycz and J.F. Peters (eds.). *Computational Intelligence in Software Engineering*. World Scientific, 3-32, 1998.

[30] McDonald B.H., and Wexler, A. "Finite Element Solution of Unbounded Field Problems". *IEEE Transactions on Microwave Theory and Techniques*, MTT-20, 12, 1972.

[31] Mikhlin, S.G. *Variational Methods in Mathematical Physics*. Oxford, Pergamon Press, 1965.

[32] Minium, E.W., Clarke, R.B., Coladarci, T. *Elements of Statistical Reasoning*. Wiley, New York, 1999.

[33] Mitchell, T.M. *Machine Learning*. McGraw-Hill, New York, 1997.

[34] Nahamias, S. *Production and Operations Analysis*. 2nd ed., Irwin, 1993

[35] National Institute of Standards & Technology. "The Economic Impacts of Inadequate Infrastructure for Software Testing". *Planning Report* 02-3, May 2002.

[36] Pfleeger, S.L. *Software Engineering: Theory and Practice*. 2nd Edition, Prentice-Hall, 2001.

[37] Quinlan, J. R. C4.5: *Programs for Machine Learning*. Morgan Kaufmann, 1993.

[38] Rao, C.R., and Toutenburg, H. *Linear Models: Least Squares and Alternatives*. Springer-Verlag, 1995.

[39] Schroeder P. J., and Korel, B. "Black-Box Test Reduction Using Input-Output Analysis". *Proc. of ISSTA* '00, 173-177, 2000.

[40] Vanmali, M., Last, M., Kandel, A. "Using a Neural Network in the Software Testing Process". *International Journal of Intelligent Systems*, 17, 1, 45-62, 2002.

[41] Voas J. M., and McGraw, G. *Software Fault Injection: Inoculating Programs against Errors*. Wiley, 1998.

[42] Weyuker, E., Goradia, T., and Singh, A. "Automatically Generating Test Data from a Boolean Specification". *IEEE Transactions on Software Engineering*, 20, 5, 353-363, 1994.

[43] Zienkiewicz, O.C. *The Finite Element Method in Engineering Science*. London, McGraw-Hill, 1971.

CHAPTER 3

AUTOMATED GUI REGRESSION TESTING USING AI PLANNING

Atif M. Memon

Department of Computer Science &
Fraunhofer Center for Experimental Software Engineering,
University of Maryland,
College Park, MD 20742, USA
E-mail: atif@cs.umd.edu

A software's use of a **graphical user interface (GUI)** can significantly raise its cost of regression testing both because GUI software is modified and retested frequently and special characteristics of GUIs, such as *event-driven input* and *graphical output* prevent the application of automated regression testing techniques to GUIs. We demonstrate that a test suite originally used to test a GUI forms two partitions for the modified GUI: *affected* test cases that *cannot* be executed on the modified GUI and *unaffected* test cases that *need not* be executed. We present a novel technique for automated GUI regression testing using **AI planning**. We represent GUI test cases at a high level of abstraction using **tasks** (*pairs of initial and goal states*). We apply planning to regenerate affected test cases from these tasks and use them for retesting. Our traditional representation of GUI test cases as sequences of events and expected states already contains the necessary mechanism to associate a task with each test case. The planning technique also blends naturally with our test case generator that employs hierarchical planning to generate test cases.

1. Introduction

An increasingly important but expensive part of a software's development process is its maintenance. Maintenance activities account for as much as two-thirds of the overall cost of software production [34, 29, 37]. *Regression testing*, a necessary maintenance activity, is performed on modified software to provide confidence that (1) the software behaves correctly, and (2) modifications have not adversely affected the software's quality. Regression testing can account for as much as one-half of the cost of software maintenance [3]. A software's use of a *graphical user interface* (**GUI**) can significantly raise its cost of regression testing. GUIs constitute as much as half of the software's code [19, 26]. GUI software requires frequent retesting because GUIs are typically designed using rapid prototyping [26], in which the GUI software is modified and tested on a continuous basis. Special characteristics of GUIs, such as *event-driven input* and *graphical output*, present a new set of challenges to the problem of GUI regression testing, requiring the development of new solutions.

Although regression testing of conventional software has received a lot of attention [6, 32, 34, 35], there has been almost no reported research on GUI regression testing. The exception is White [40] who proposes a Latin square method to reduce the size of the regression test suite. The underlying assumption is that it is enough to check pair-wise interactions between menu-items of the GUI. The technique requires that each menu-item appear in at least one test case. This strategy seems promising since it also employs GUI events. However, the technique needs to be extended to GUI items other than menus. Moreover, detailed studies need to be conducted to verify whether the pair-wise interactions checking assumption is sufficient.

Several strategies for regression testing of conventional software have been proposed [2, 9, 30, 17]. One regression testing strategy proposes rerunning all test cases that have not become obsolete. Since this *retest-all* strategy is resource intensive, numerous efforts have been made to reduce its cost. *Selective retest techniques* [1, 4, 11] attempt to reduce the cost of regression testing by testing only selected parts of the software. These techniques have traditionally focused on two problems: (1)

regression test selection problem, i.e., selecting a subset of the existing test cases [34], and (2) *coverage identification problem*, i.e., identifying portions of the software that require additional testing. Solutions to the regression test selection problem traditionally compare structural representations (e.g., *control-flow graphs* [34], *control-dependence graphs* [33]) of the original and modified software. Test cases that cause the execution of different paths in these structures are likely to be selected for re-testing. Among selective retest strategies, the *safe* approaches require the selection of every existing test case that exercises any program element that could be affected by a given program change. Although computationally less expensive than the retest-all strategy, safe approaches still make heavy demands on resources. At the other end of the spectrum of selective retest strategies are minimization approaches that attempt to select the smallest set of test cases necessary to test affected program elements at least once [36]. These techniques attempt to assure that some structural coverage criterion is met by the test cases that are selected. Practical strategies fall between the safe strategies and minimization strategies. The test designer may be satisfied with using near-minimal sets of test cases [31].

Other regression testing techniques include analyzing changes in *functions*, *types*, *variables*, and *macro definitions* [30], using *def-use chains* [9], constructing *procedure dependence graphs* [5], and analyzing *code* and *class hierarchy* for object-oriented programs [17]. These techniques are not directly applicable to GUI regression testing because regression information is derived from changes made to the software's code. However, if a logical structure of the user event sequences can be constructed, then some of the ideas from these techniques may be applicable.

Selective retest techniques for traditional software attempt to reduce the cost of regression testing by identifying *portions* of the modified software that should be retested. The original test suite is partitioned into (1) test cases that do not have to be rerun on the modified software because they are not *modification-traversing*, (2) test cases that have to be rerun because they are modification-traversing, and (3) *obsolete* test cases that are no longer needed to test the software [34, 33]. New test

cases may also be generated to test those parts of the modified software not tested by (2) above. The above partitioning of original test cases cannot be applied to GUIs because the structure of a GUI test case, given below, only allows cases (1) and (3) from above; that is, no test case from the original test suite can/should be rerun.

A GUI test case consists of 3 parts: (a) an *initial GUI state* S_0 in which the test case is executed, (b) a *sequence of events* e_1; e_2; ...; e_n which is the test input to the GUI, and (c) *a sequence of states* S_1; S_2; ...; S_n, where S_i is the expected state of the GUI after event e_i is executed [21]. The initial state is used to initialize the GUI to a desired state before events are executed on it and the expected state sequence is used by *test oracles* [23] to determine whether the GUI executed correctly during testing. Oracles are necessary when testing GUIs because incorrect behavior at any one state can prohibit executing the rest of the test case. A change in the GUI may render some of the test cases useless, either because they specify *an unreachable initial state*, *incorrect sequence of events*, or *incorrect expected states*[1] for the modified GUI. Such test cases are called *affected* and cannot be executed on the modified GUI. The remaining test cases (*unaffected*) execute exactly the same sequence of unchanged events on the modified GUI as they did on the original GUI and specify correct expected state. These test cases are not likely to reveal faults in the modified GUI since the GUI has already been tested for these sequences.

Since the affected test cases *cannot* be executed on the GUI and the unaffected test cases *need* not be rerun, none of the original test cases can be reused during GUI regression testing. (We provide more details in Section 2 and specific examples in Section 3). Consequently new test cases have to be generated from scratch to retest the GUI. Since the purpose of regression testing is to retest pre-tested parts of the software,

[1] Note that a valid test case should contain an accurate description of the *intended* expected state for the modified GUI. If a test case with an incorrect expected state is executed, then the test case would fail for a correctly executing GUI, hence resulting in a false positive that must be manually identified. Our goal here is to eliminate false positives by not executing test cases that contain incorrect expected states in the first place.

the GUI test designer must keep track of all such parts and regenerate test cases to target these parts. Since the analysis required for such selective regeneration is expensive, in practice no regression testing of GUIs is performed.

In this paper, we present a novel technique to perform GUI regression testing. We represent GUI test cases at a high level of abstraction using *tasks* (pairs of initial and goal states). Even when changes to the GUI make GUI test cases unusable, these tasks remain valid across successive GUI versions. We employ *planning* to regenerate affected test cases from these tasks and use them for retesting. The use of tasks is in line with the philosophy of regression testing; after all the purpose of regression testing is to reaffirm that a software's functionality did not unintentionally break across its versions. Tasks provide a simple and intuitive way to represent a GUI's functionality. Our traditional representation of GUI test cases, as sequences of events and expected states (discussed in detail in Section 4) already contains the necessary mechanism to associate a task with each test case. The planning technique also fits naturally with our test case generator that employs hierarchical planning to generate test cases [22, 24]. Since our primary goal is automation, we use a GUI model to automatically detect modifications to the GUI and identify test cases that need to be rerun. We also use the original test cases as a cache to reuse its useful parts. We have performed experiments on the popular **xfig**[2] software that show that our regression testing techniques are practical.

We have implemented our techniques to develop an automated regression tester. During its development, we faced several challenges: First, we had to automatically identify parts of the GUI that changed so that they could be used to identify test cases to rerun. Second, we had to represent the test cases so that they did not become completely unusable when changes were made to the GUI. Finally, we had to develop a mechanism to give the test designer a way to represent the GUI's functionality as it changed. In this paper, we present details of these challenges and the design of the regression tester.

---

[2] Xfig is a drawing tool available from `http://www.xfig.org`.

In the next section, we describe what we mean by affected and unaffected test cases. In Section 3 we give an overview of the design of the regression tester and our re-planning technique by using an example. In Section 4, we formally define a GUI's model and a GUI test case. The detailed design of the regression tester is described in Section 5. We then describe experiments and their results in Section 6. Finally, we conclude in Section 7 with a discussion of ongoing and future work.

2. Affected and Unaffected Test Cases

Since a GUI test case consists of a reachable initial state S_0, a legal event sequence e_1; e_2; ...; e_n, and expected states S_1; S_2; ...; S_n, a modification to the GUI may affect *any* (or *all*) of these parts. For example, a change to the structure of the GUI may cause a test case's event sequence to become *illegal*. A modification to the semantics of an event in the GUI may make the expected state part of the test case *incorrect*. Neither of these test cases can be executed on the modified GUI.

Table 1: All Possible Effects of GUI Modifications to the Parts of a Test Case.

row	**initial state** S_0	**event sequence** e_1; e_2; ...; e_n	**expected state** S_1; S_2; ...; S_n	**test case status**
1	reachable	legal	correct	unaffected
2	unreachable	x	x	affected
3	reachable	illegal	x	affected
4	reachable	legal	incorrect	affected

Table 1 shows all the possible ways in which modifications made to a GUI may affect the three parts of a test case. The second to the fifth columns represent the effects of modifications to the initial state, event sequence, expected state, and test case respectively. The first row shows the case where a test case was not affected by the GUI modifications, since its initial state is reachable, it has a legal event sequence and a corresponding correct expected state. Such a test case is called an *unaffected* test case. Unaffected test cases need not be run on the

modified GUI since they will re-execute a sequence of unmodified events that have already been tested on the original GUI.

The second row shows the case where the initial state of the test case became unreachable. Entries marked with "*x*" indicate "*don't care*" conditions, i.e., if the initial state of a test case is unreachable, it does not matter whether the event sequence is legal/illegal or the expected state is correct/incorrect – the test case cannot be executed. The third row shows that the GUI modification changed the structure of the GUI, causing the event sequence part of the test case to become illegal. Test cases of rows 2 and 3 cannot be executed on the GUI. Finally, the fourth row shows that a modification caused the expected state part of the test case to become incorrect because the semantics of one of the events in the test case's event sequence changed. Although the event sequence of this test case is legal, its corresponding expected state cannot be used to verify the correctness of the GUI as it executes. Executing such a test case is not useful since the tester cannot determine whether or not the GUI executed correctly. One possible approach is to inform the test designer of this change so that the expected state could be updated manually; we provide a mechanism to update the expected state automatically. Note that a test case with an incorrect expected state is unable to detect both intended and/or unintended modifications to the GUI. Since the test case does not encode correct intended behavior, it will always flag an error as long as the GUI exhibits behavior that is different from the original version. For correct operation, the test case must contain an accurate description of intended GUI behavior.

As the table shows, test cases represented by rows 2, 3, and 4 are called affected test cases. Although an affected test case cannot be executed on the GUI, it contains valuable information in the event sequence and expected states about how the modifications have changed the execution behavior of the GUI.

3. Overview

Before we present the details of the GUI model and regression testing algorithms, in this section we first present an intuitive overview of the regression testing process by using an example. Figure 1 shows a high-level view of the regression tester. The inputs are the *original test suite*, generated to test the original GUI, and *representations* of both the original and modified GUIs. The outputs are the *test cases* that need to rerun, *unaffected test cases* that need not be rerun, and *discarded* test cases. The key components of the regression tester include:

- **Test case selector** that partitions the original test suite into (1) unaffected test cases, (2) test cases for obsolete tasks, e.g., those that contain an unreachable initial state, (3) test cases that are affected because they specify an illegal event sequence for the modified GUI, and (4) test cases that are affected because they specify incorrect expected state for the modified GUI.
- **Planning-based test case regenerator** that uses planning to regenerate the affected test cases that have an illegal event sequence. If successful, then the regenerated test case is used for regression testing; otherwise, if the planner fails to find a plan, then the task is obsolete and the test case is discarded.
- **Expected-state regenerator** that regenerates the expected states for test cases. In case it is unable to regenerate the expected state, then the task is obsolete and the test case is discarded.

Details of the design of each of the above components is presented in Section 5. We now give an overview of their operation by taking an example of the popular *xfig* software available on most Unix platforms. We use versions *Xfig 3.2 patchlevel 0-beta4 (Protocol 3.2)* and *Xfig 3.2 patchlevel 3c (Protocol 3.2)*, which we will refer to as the *original* and *modified* software respectively. The original software is shown in Figure 2. Note that the shaded boxes are not a part of the GUI. They represent labels that we have used to represent events in the GUI. Dashed lines show the relationships between these names and the corresponding events. For example, the shaded box **TypeInTextField** represents an event used to enter text in the text-field. Solid line arrows show the

relationships between windows and the events used to open them. For example, the event **File** invokes the window entitled "**Xfig: File menu**".

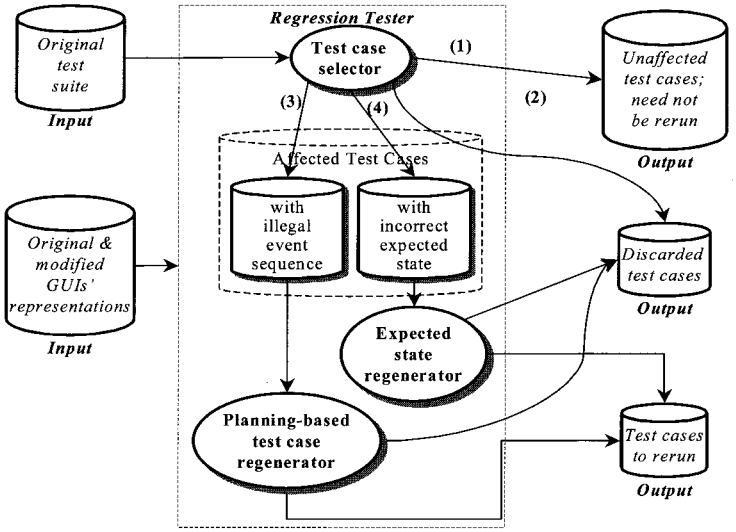


Figure 1. A High-level Overview of the Regression Tester.

Consider the test case, generated for the original software, shown in Figure 3. The test case uses a new event **ClickOnObject(OBJECT)**, which translates to clicking the left mouse button on an **OBJECT**. Note that we have represented events using the function notation with parameters. The exact representation details are discussed later in Section 4. The test case first launches **xfig**, loads a file named **original.fig**, cuts a circle object, saves the resulting drawing in **new.fig**, and exits the application. An example of running this test case using a specific instance of the file **original.fig** is shown in Figure 4.

Now consider the modified software shown in Figure 5. The modified software contains most of the functionality of the original software. Specifically, a user is able to load/save files and delete objects. A test designer performing regression testing on the modified software would need to verify whether the tested functionality that was available in the original software works correctly in the modified software. However, examining the test case of Figure 3. shows that it cannot even be

executed on the modified software; performing **File** opens a *pull-down menu*, (not the window as originally done) preventing event **TypeInTextField** from being performed. Hence the modifications have made the test case of Figure 3 useless for the modified software since the event sequence of the test case has become illegal. The test case selector marks this test case as *affected* so that the planning-based test case regenerator can regenerate it. However, from our prior experience of using **xfig**, we know that the modified software *can* be used to perform the task of Figure 4. If we were to perform the task manually, we would need to adapt the event sequence to the modified software. The modified test case needed to achieve the same task using the modified software is shown in Figure 6. Our regression tester uses planning to generate the modified test case automatically. This test case can be used for regression testing.

In principle, the approach outlined above can be used for regression testing of GUIs if tasks are maintained with each test case. However, regenerating test cases from scratch is unnecessary since parts of the original test case are still valid and may be reused. We represent the GUI as a hierarchy of *components* and employ a form of hierarchical planning with *caching* to reuse these parts. We provide details of this technique in Section 5.

The entire testing process can be partitioned into three major phases. In the first **Setup** phase the automated system creates a GUI model consisting of *operators* for planning, *event-flow graphs*, and an *integration tree* [25] that is used throughout the testing process. Then the test designer defines the *preconditions* and *effects* of each operator. The test designer starts the **Test Case Generation** phase by identifying *tasks*. These tasks are used by a test case generator to automatically generate test cases. The test designer modifies the GUI as the first step of the **Regression Testing** phase. The automated system automatically updates the GUI model, identifies test cases that need to be rerun and generates the regression test suite. This process is summarized in Table 2

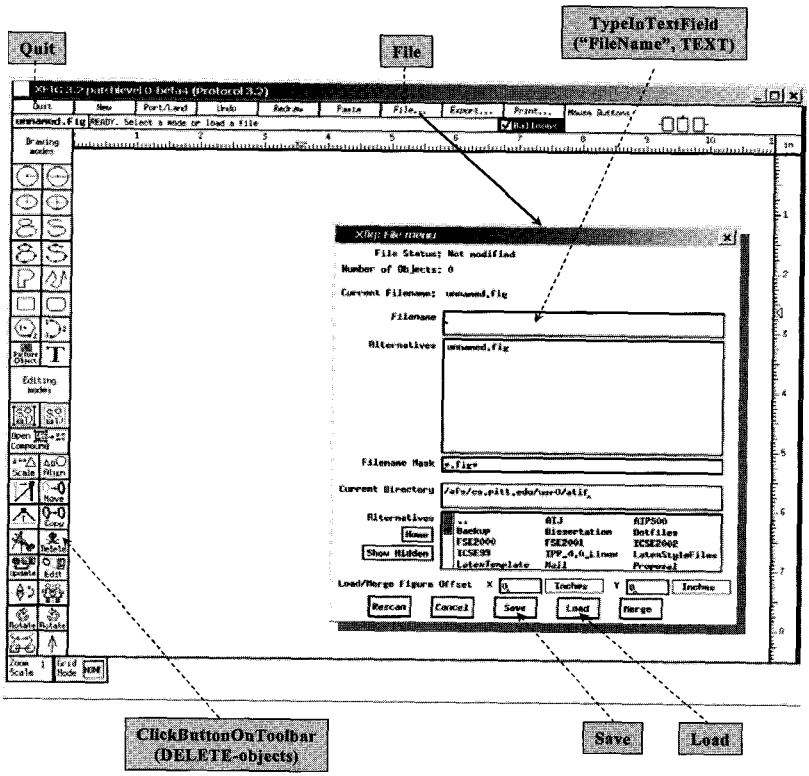


Figure 2. The Original Software (Xfig 3.2 patchlevel 0-beta4 (Protocol 3.2))

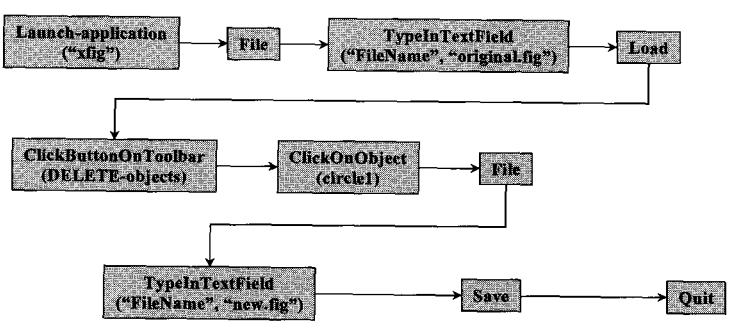


Figure 3. A Test Case for the Original Software

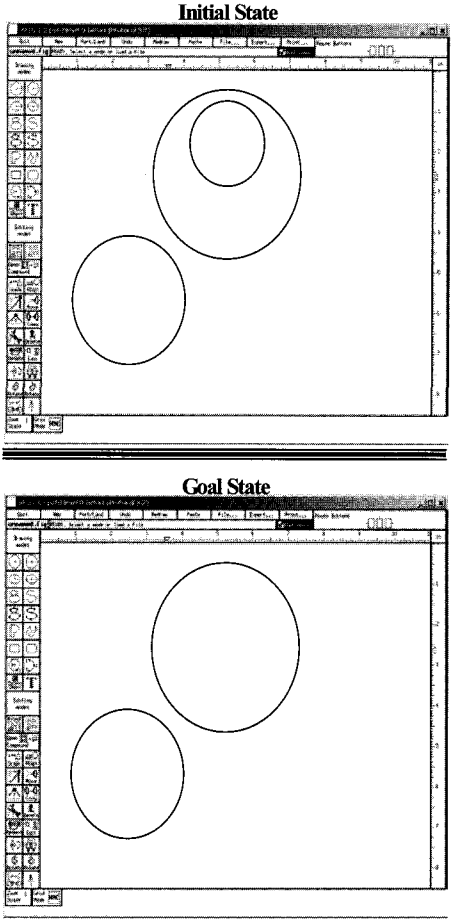


Figure 4. Running the Test Case on a Specific Instance of original.fig

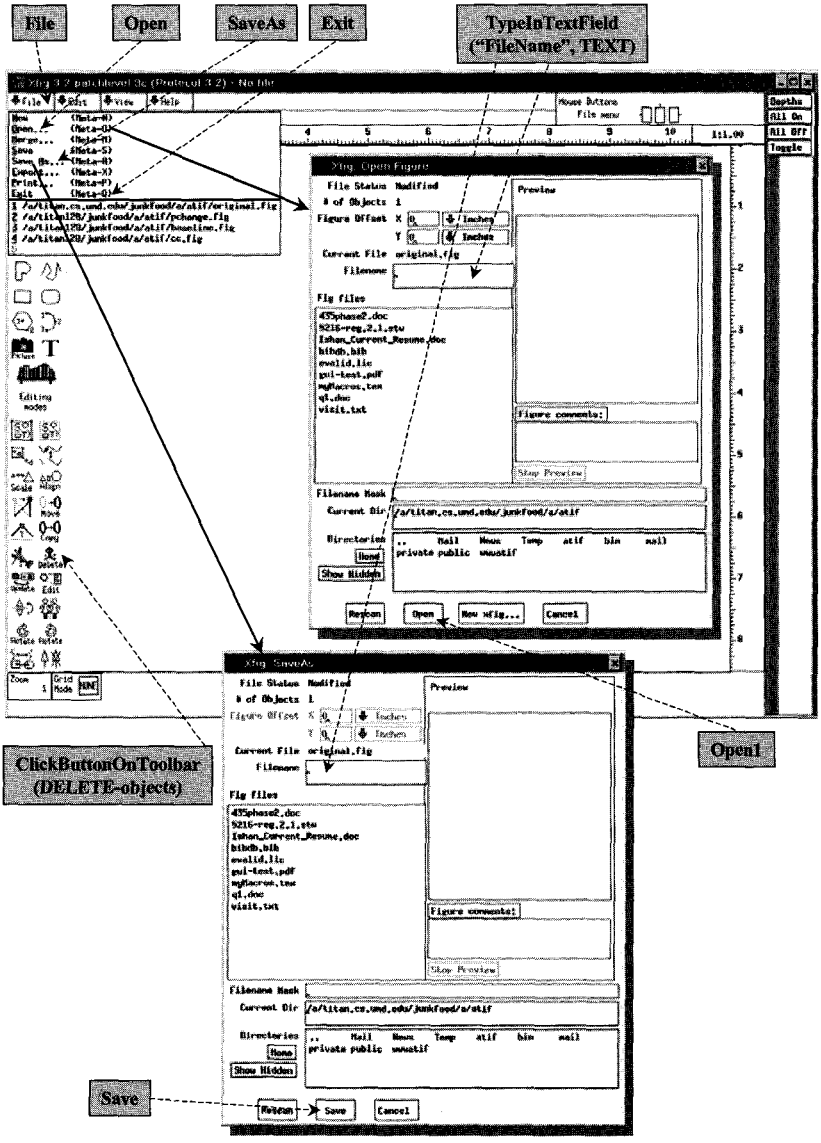


Figure 5. The Modified Software (Xfig 3.2 patchlevel 3c (Protocol 3.2))

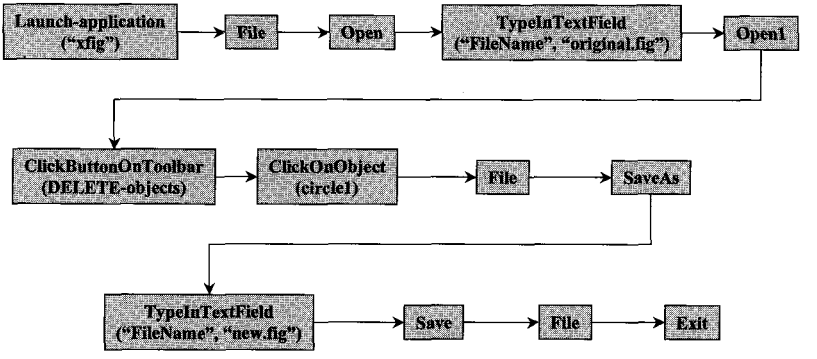


Figure 6. A Test Case for the Modified GUI

Table 2: Roles of the Test Designer and Automated System During Regression Testing.

Phase	Step	Test Designer	Planning-based System
Setup	1		Derive Hierarchical GUI Operators, Event-flow Graphs, Integration Trees
	2	Define Preconditions and Effects of Operators	
Test-case Generation	3	Identify Tasks	
	4		Generate Test Cases
Regression Testing	5	Modify GUI Design	
	6		Detect Changes to GUI Model
	7		Identify Affected Test Cases that Need to be Rerun
	8		Generate Regression Test Suite

4. Representation

We now present a formal model of a GUI and a GUI test case. A GUI test case consists of a reachable initial GUI state S_0, a legal sequence of events e_1; e_2; ...; e_n, and a correct sequence of states S_1; S_2; ...; S_n. This section provides precise definitions of these terms.

4.1 Representing the GUI's State

A GUI's state is modeled as a set of *objects*, (**label**, **form**, **button**, **text**, etc.) and a set of *properties* of those objects (**background-color**, **font**, **caption**, etc.). Each GUI will use certain types of objects with associated properties; at any specific point in time, the GUI can be described in terms of the specific objects that it contains and the values of their properties.

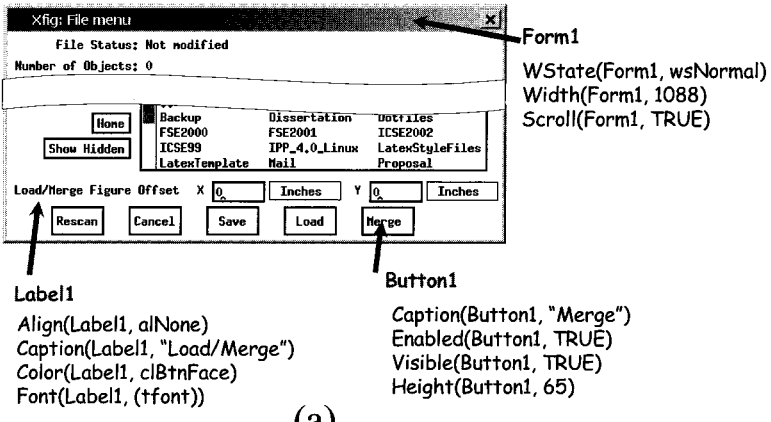
Formally, a GUI is modeled at a particular time t in terms of:

- its **objects** $O = \{o_1, o_2, ..., o_m\}$, and
- the **properties** $P = \{p_1, p_2, . . . , p_l\}$ of those objects. Each property p_i is an n_i-ary Boolean relation, for $n_i \geq 1$, where the first argument is an object $o_1 \in O$. If $n_i > 1$, the last argument may either be an object or a property value, and all the intermediate arguments are objects. The (*optional*) property value is a constant drawn from a set associated with the property in question: for instance, the property "**background-color**" has an associated set of values, {**white**, **yellow**, **pink**, etc.}. A distinguished set of properties, the object types, which are unary relations, ("**window**", "**button**") is assumed to be available. A common example is the **Cancel** button, one of whose properties is called **Caption** and its current value is "**Cancel**".

There are several points that should be noted about the description of properties. First, properties are relations, not functions, and so there may sometimes be multiple values for the same property of a given object. For example, there may be multiple objects in a window. Next, properties as defined are fluents [20], i.e., relations that are true in some situations (or states of the world) and not others. An everyday example of a fluent is the relation **president(US, Bush)**, with the obvious meaning, where the state it is evaluated in is the state of the real world. The fluents are evaluated with respect to a state of the GUI. Finally, a fluent may be undefined in some states, for example, **president(US, Dole)** in the state of the world in the year 1567, or **background-color(w24, blue)** in the state of a GUI immediately after window **w24** has been destroyed.

Definition: The ***state*** of a GUI at a particular time t is the set P of all the properties of all the objects O that the GUI contains.

A description of the state would contain information about the types of all the objects currently extant in the GUI, as well as all of the properties of each of those objects. For example, consider the **Xfig: File menu** GUI shown in Figure 7(a). This GUI contains several objects, three of which are explicitly labeled; for each, a small subset of its properties is shown. The state of the GUI, partially shown in Figure 7(b), contains all the properties of all the objects in **Xfig: File menu**. Note that our definition of a GUI state is general and may be used to model GUIs as finite state automata or in state chart diagrams. We use the state in a unique way by modeling the event structure of the GUI, presented next.



(a)

State = {Align(Label1, alNone), Caption(Label1, "Load/Merge"), Color(Label1, clBtnFace), Font(Label1, (tfont)), WState(Form1, wsNormal), Width(Form1, 1088), Scroll(Form1, TRUE), Caption(Button1, "Merge"), Enabled(Button1, TRUE), Visible(Button1, TRUE), Height(Button1, 65), ...}

(b)

Figure 7. (a)The Xfig: File menu GUI with three objects explicitly labeled and their associated properties, and (b) the State of the Xfig: File menu GUI

4.2 Representing GUI Events

The state of a GUI is not static; events preformed on the GUI change its state. Events are modeled as functions from one state of the GUI to another.

Definition: The ***events*** $E = \{e_1, e_2, ..., e_n\}$ associated with a GUI are functions from one state of the GUI to another state of the GUI.

Since events may be performed on different types of objects, in different contexts, yielding different behavior, they are parameterized with objects and property values. For example, an event **set-background-color(*w*, *x*)** may be defined in terms of a **window w** and **color x**; **w** and **x** may take specific values in the context of a particular GUI execution. As shown in Figure 8, whenever the event **set-background-color(w19, yellow)** is executed in a state in which window **w19** is open, the background color of **w19** should become yellow (or stay **yellow** if it already was), and no other properties of the GUI should change. This example illustrates that, typically, events can only be executed in some states; **set-background-color(w19, yellow)** cannot be executed when window **w19** is not open.

It is of course infeasible to give exhaustive specifications of the state mapping for each event: in principle, as there is no limit to the number of objects a GUI can contain at any point in time, there can be infinitely many states of the GUI. Hence, GUI events are represented using operators, which specify their preconditions and effects:

Definition: An ***operator*** is a 3-tuple <**Name**, **Preconditions**, **Effects**> where:

- **Name** identifies an event and its parameters.
- **Preconditions** is a set of positive ground literals $p(arg_1, ..., arg_n)$, where p is an n-ary property (i.e., $p \in P$). *Pre(Op)* represents the set of preconditions for operator *Op*. An operator is applicable in any state S_i in which all the literals in *Pre(Op)* are **true**.

- **Effects** is also a set of positive or negative ground literals $p(arg_1, ..., arg_n)$, where p is an n-ary property (i.e., $p \in P$). *Eff(Op)* represents the set of effects for operator *Op*. In the resulting state S_j, all of the positive literals in *Eff(Op)* will be true, as will all the literals that were true in S_i except for those that appear as negative literals in *Eff(Op)*.

For example, the following operator represents the **set-background-color** event discussed earlier:

Name: set-background-color(wX: window, Col: Color)
Preconditions: is-current(wX), background-color(wX, oldCol), oldCol != Col
Effects: background-color(wX, Col)

Going back to the example of the GUI in Figure 8 in which the following properties are true before the event is performed: **window (w19), background-color (w19, blue), is-current (w19)**. Application of the above operator, with variables bound as **set-background-color(w19, yellow)**, would lead to the following state: **window(w19), background-color (w19, yellow), is-current (w19)**, i.e., the background color of window **w19** would change from **blue** to **yellow**.

The above scheme for encoding operators is the same as what is used in the AI planning literature [27, 38, 39]; the persistence assumption built into the method for computing the result state is called the "STRIPS assumption". A complete formal semantics for operators making the STRIPS assumption has been developed by Lifschitz [18]. Given that GUI specifications can describe the intended behavior of events in terms of their preconditions and effects [8, 7], it is relatively straightforward for the test designer to construct operators for the GUI model.

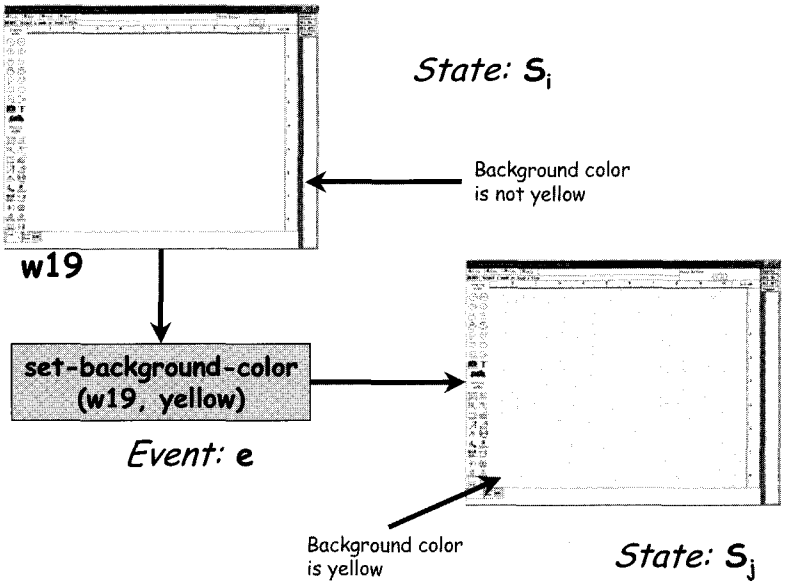


Figure 8. An Event Changes the State of the GUI.

One final point to note about the representation of effects is the inability to efficiently express complex events when restricted to using only sets of literals. Although in principle, multiple operators could be used to represent almost any event, complex events may require the definition of an exponential number of operators, making planning inefficient [13]. In practice, a more powerful representation that allows conditional and universally quantified effects is employed. For example, the operator for the **Paste** event would have different effects depending on whether the clipboard was empty or full. Instead of defining two operators for these two scenarios, a conditional effect could be used. In cases where even conditional and quantified effects are inefficient, procedural attachments, i.e., arbitrary pieces of code that perform the computation, are embedded in the effects of the operator [12]. One common example is the representation of computations. A calculator GUI that takes as input two numbers, performs computations (such as addition, subtraction) on the numbers, and displays the results in a text

field will need to be represented using different operators, one for each distinct pair of numbers. By using a procedural attachment, the entire computation may be handled by a piece of code, embedded in a single operator.

4.3 Representing Executable Event Sequences

In this section, the representation of an event in terms of its preconditions and effects is used to develop a formal representation of an executable event sequence. The function notation $S_j = e(S_i)$ is used to denote that S_j is the state resulting from the execution of event e in state S_i. Events can be strung together into sequences.

> **Definition**: $e_1 \circ e_2 \circ \ldots \circ e_n$ is an ***executable event sequence*** for a state S_0 iff there exists a sequence of states $S_0; S_1; \ldots ; S_n$ such that $S_i = e_i(S_{i-1})$, for $i = 1, \ldots, n$.

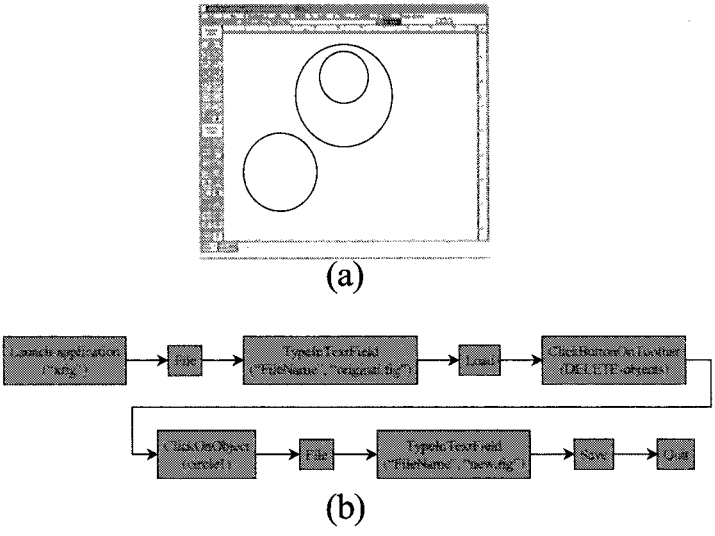


Figure 9. (a) A State S_0 for xfig and (b) an Executable Event Sequence for S_0

Figure 9 shows **xfig** in a state S_0 and an executable event sequence corresponding to S_0. Extending the function notation above, $S_j = (e_1 \circ e_2 \circ \ldots \circ e_n)(S_i)$, where $e_1 \circ e_2 \circ \ldots \circ e_n$ is an executable event sequence, denotes that S_j is the state that results from executing the specified sequence of events starting in state S_i.

With each GUI is associated a distinguished set of states called its valid initial states.

Definition: A set of states S_I is called the ***valid initial state set*** for a particular GUI iff the GUI may be in any state $S_i \in S_I$ when it is first invoked.

Given a GUI in state $S_i \in S_I$, i.e., in a valid initial state of the GUI, new states may be obtained by performing events on S_i. These states are called the reachable states of the GUI. Formally, a reachable state is defined as follows.

Definition: The state S_j is a ***reachable state*** iff either $S_j \in S_I$ or there exists an executable event sequence $e_x \circ e_y \circ \ldots \circ e_z$ such that $S_j = (e_x \circ e_y \circ \ldots \circ e_z)(S_i)$, for any $S_i \in S_I$.

4.4 GUI Components and Event Classification

Since today's GUIs are large and contain a large number of events, any scalable representation must decompose a GUI into manageable parts. As mentioned previously, GUIs are hierarchical, and this hierarchy may be exploited to identify groups of GUI events that can be analyzed in isolation. One hierarchy of the GUI and the one used in this research is obtained by examining the structure of modal windows in the GUI.

Definition: A ***modal*** window is a GUI window that, once invoked, monopolizes the GUI interaction, restricting the focus of the user to a specific range of events within the window, until the window is explicitly terminated.

The **Xfig: File menu** window is an example of a modal window in **xfig**. As Figure 10 shows, when the user performs the event **Open**, a window entitled **Xfig: File menu** opens and the user spends time selecting the file, and finally explicitly terminates the interaction by either performing **Load**, **Cancel**, etc.

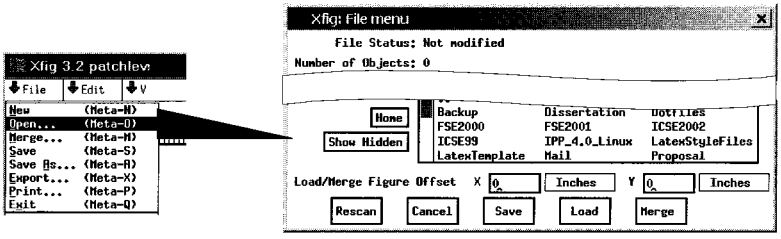


Figure 10. The Event Open Opens a Modal Window

Other windows in the GUI are called *modeless* windows that do not restrict the user's focus; they merely expand the set of GUI events available to the user. For example, in the **xfig** software, performing the event **Search/Replace** opens a modeless window entitled **Xfig: Search & Replace** (Figure 11).

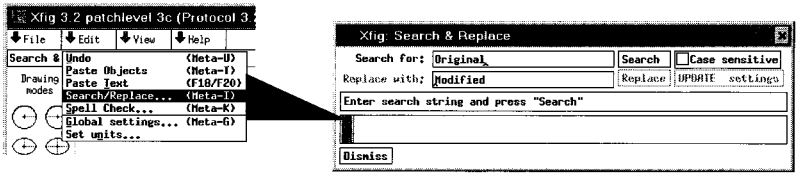


Figure 11. The Event Search/Replace Opens a Modeless Window

At all times during interaction with the GUI, the user interacts with events within a modal dialog. This modal dialog consists of a modal window **X** and a set of modeless windows that have been invoked, either directly or indirectly by **X**. The modal dialog remains in place until **X** is explicitly terminated. Intuitively, the events within the modal dialog form a GUI component.

Definition: A GUI ***component*** *C* is an ordered pair (*RF, UF*), where *RF* represents a modal window in terms of its events and *UF* is a set whose elements represent modeless windows also in terms of their events. Each element of *UF* is invoked either by an event in *UF* or *RF*.

Note that, by definition, events within a component do not interleave with events in other components without the components being explicitly invoked or terminated.

Since components are defined in terms of modal windows, a *classification* of GUI events is used to identify components. The classification of GUI events is as follows:

- *Restricted-focus* events open modal windows. Open in Figure 10 is a restricted-focus event.
- *Unrestricted-focus* events open modeless windows. For example, Search/Replace in Figure 11is an unrestricted-focus event.
- *Termination* events close modal windows; common examples include Load, Save, and Cancel (Figure 10).

The GUI contains other types of events that do not open or close windows but make other GUI events available. These events are used to open menus that contain several events.

- *Menu-open events* are used to open menus. They expand the set of GUI events available to the user.

Menu-open events do not interact with the underlying software. Note that the only difference between menu-open events and unrestricted-focus events is that the latter open windows that must be explicitly terminated. The most common example of menu-open events are generated by buttons that open pull-down menus. For example, in Figure 11, **File**, **Edit**, **View**, and **Help** are menu-open events.

Finally, the remaining events in the GUI are used to interact with the underlying software.

- *System-interaction events* interact with the underlying software to perform some action; common examples include the Cut, Copy, and Paste events commonly used for moving objects to/from the clipboard.

4.5 Event-flow Graphs

A GUI component may be represented as a flow graph. Intuitively, an event-flow graph represents all possible interactions among the events in a component.

Definition: An ***event-flow graph*** for a component C is a 4-tuple $<V, E, B, I>$ where:

1. V is a set of vertices representing all the events in the component. Each $v \in V$ represents an event in C.
2. $E \subseteq V \times V$ is a set of directed edges between vertices. Event e_i **follows** e_j iff e_j may be performed immediately after e_i. An edge $(v_x, v_y) \in E$ iff the event represented by v_y follows the event represented by v_x.
3. $B \subseteq V$ is a set of vertices representing those events of C that are available to the user when the component is first invoked.
4. $I \subseteq V$ is the set of restricted-focus events of the component.

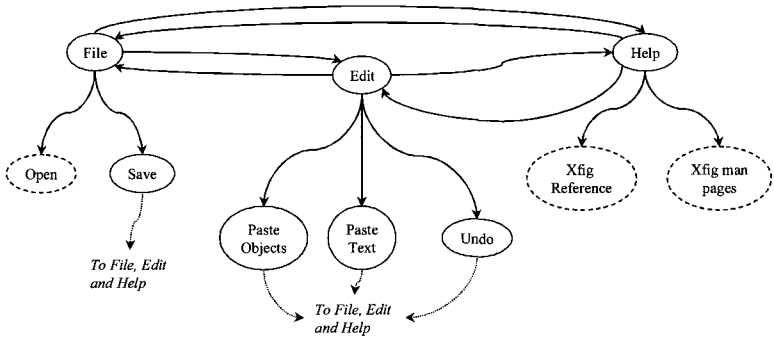


Figure 12. Event flow Graph for the Main Component of xfig

An example of an event-flow graph for a part of the **Main** component of **xfig** is shown in Figure 12. At the top are three vertices (**File**, **Edit**, and **Help**) that represent part of the pull-down menu of **xfig**. They are events that are available when the **Main** component is first invoked.

Once **File** has been performed in **xfig**, any of **Edit**, **Help**, **Open**, and **Save** events may be performed. Hence there are edges in the event-flow graph from **File** to each of these events. Note that **Open**, **Xfig Reference** and **Xfig man pages** are shown with *dashed* ovals. We use this notation for events that invoke other components, i.e., I = {**Open**, **Xfig Reference**, **Xfig man pages**}. Other events include **Save**, **Paste objects**, **Paste text**, and **Undo**. After any of these events is performed in **xfig**, the user may perform **File**, **Edit**, or **Help**, shown as edges in the event-flow graph.

4.6 Integration Tree

Once all the components of the GUI have been represented as event-flow graphs, the remaining step is to identify interactions among components. A structure called an integration tree is constructed to identify interactions (*invocations*) among components.

Definition: Component C_x ***invokes*** component C_y if C_x contains a restricted-focus event e_x that invokes C_y.

Intuitively, the integration tree shows the **invokes** relationship among all the components in a GUI. Formally, an integration tree is defined as:

Definition: An ***integration tree*** is a 3-tuple $< N, R, B >$, where N is the set of components in the GUI and $R \in N$ is a designated component called the **Main** component. B is the set of directed edges showing the **invokes** relation between components, i.e., $(Cx, Cy) \in B$ iff Cx **invokes** Cy.

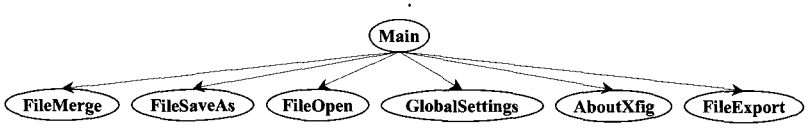


Figure13. An Integration Tree for a Part of xfig

Figure 13 shows an example of an integration tree representing a part of the xfig's GUI. The nodes represent the components of the xfig GUI and the edges represent the invokes relationship between the components. The tree in Figure 13 has an edge from Main to FileOpen showing that Main contains an event, namely Open (see Figure 12) that invokes FileOpen.

4.7 Representing GUI Test Cases

To test a GUI, event sequences for the GUI must be executed.

> **Definition**: A ***legal event sequence*** of a GUI is $e_1; e_2; e_3; \ldots; e_n$ where either $(e_i, e_{i+1}) \in E$, for some component of the GUI, or e_i is a restricted-focus event that invokes component C_x and e_{i+1} is an event in C_x, for $1 \leq i \leq n - 1$.

Note that a legal event sequence is less restricted than an executable event sequence. Hence the set of all legal event sequences also contains all executable event sequences. During testing, it is important to test legal sequences, even though they may not all be executable.
A formal representation of a GUI test case is as follows:

> **Definition**: A ***GUI test case*** T is a triple $<S_0, e_1; e_2; \ldots; e_n, S_1; S_2; \ldots; S_n>$, consisting of a reachable state S_0, called the *initial state* for T, a legal event sequence $e_1; e_2; \ldots; e_n$ for S_0, and expected states $S_1; S_2; \ldots; S_n$, where $S_i = e_i(S_{i-1})$ for $i = 1, \ldots, n$.

Note that S_0 and S_n are already a part of the test case and represent the task associated with this test case.

5. Design of the Regression Tester

As shown earlier in Figure 1, the regression tester consists of three main components: *test case selector*, *test case regenerator*, and *expected state regenerator*. This section provides details of the design of these components.

Test case selector

The test case selector's primary function is to identify affected test cases. In addition, it performs preliminary identification of discarded test cases. The logical functionality of the test case selector is summarized as a graph in Figure 14. The nodes in the graph correspond to three parts of the test case selector that check whether a test case is affected. These components are described next.

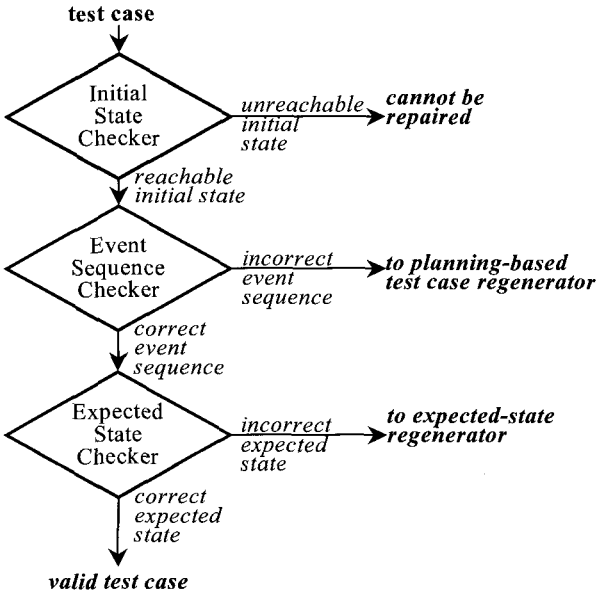


Figure 14. Parts of the Test Case Selector

5.1.1 Initial State Checker

The initial state checker determines whether the initial state S_0 associated with a test case is reachable for the modified GUI. A test case with an unreachable initial state is useless since the GUI cannot be brought into the state to execute the test case. If $S_0 \in S_I$ (the set of valid initial states of the modified GUI), then it is reachable; otherwise the checker

determines a sequence of events e_x; e_y; ...; e_z that changes a state $S_x \in S_I$ to S_0. Currently, the regression tester determines this sequence by employing planning [24]. If, for at least one $S_x \in S_I$, a sequence of events exists in the GUI to change S_x to S_0 then S_0 is a reachable initial state; otherwise it is unreachable. In case the initial state is unreachable, the test case is discarded.

5.1.2 Event-Sequence Checker

The event-sequence checker determines whether the event sequence in the test case is legal. The first step is to identify the modifications made to the GUI and their effects. The key idea is to determine the modifications made to the event-flow graphs and integration tree of the original GUI to obtain the modified GUI.

Since we want this analysis to be efficient, instead of making a general comparison of the integration trees and event-flow graphs, we simplify the analysis by making a number of reasonable assumptions. We assume that events and components (1) have unique names (renaming can be carried out to accomplish this) and (2) are not renamed across versions of the GUI unless they are modified. For example, if an event **File** is not modified, then it is called **File** in the modified GUI. In case some events or components are renamed, then the test designer is made aware of these changes by the GUI developer who must maintain a log of all such changes.

Using these assumptions, we are able to automatically identify and classify GUI modifications as simple additions and deletions to the event-flow graphs and integration trees. Note that similar approaches of tracking additions/deletions have been used for incremental data-flow and code-optimizations [28] and incremental testing [10].

The following modifications may be made to events within a component, represented by an event-flow graph:

1. a vertex may be deleted,
2. a vertex may be added,
3. an edge may be deleted, and
4. an edge may be added.

If EFG_o and EFG_m are the event-flow graphs of a component that exists in both the original GUI and the modified GUI respectively, then the following sets of modifications are obtained by performing set subtraction. Note that the functions Vertices and Edges return the sets V (the set of vertices) and E (the set of edges) for the event-flow graph in question.

1. The set of all new vertices in the event-flow graph: **vertices_added** ← **Vertices**(EFG_m) − **Vertices**(EFG_o);
2. The set of all vertices deleted from the original event-flow graph: **vertices_deleted** ← **Vertices**(EFG_o) − **Vertices**(EFG_m);
3. The set of all new edges added to the event-flow graph: **efg_edges_added** ← **Edges**(EFG_m) − **Edges**(EFG_o);
4. The set of edges deleted from the original event-flow graph: **efg_edges_deleted** ← **Edges**(EFG_o) − **Edges**(EFG_m);

The above sets can be used to identify affected test cases.

Similarly, the following changes may be made to an integration tree:

1. a component may be added,
2. a component may be deleted,
3. an edge may be added, and
4. an edge may be deleted.

Let T_o and T_m be the integration trees of the original and modified GUI respectively. The following sets of modifications may be obtained from these two integration trees. Note that **Nodes** and **CompEdges** return the sets N and B for the integration tree respectively.

1. The set of components added to the integration tree: **components_added** ← **Nodes**(T_m) − **Nodes**(T_o);
2. The set of components deleted from the integration tree: **components_deleted** ← **Nodes**(T_o) − **Nodes**(T_m);
3. The set of edges added to the integration tree: **comp_edges_added** ← **CompEdges**(T_m) − **CompEdges**(T_o);
4. The set of edges deleted from the integration tree: **comp_edges_deleted** ← **CompEdges**(T_o) − **CompEdges**(T_m);

Note the difference between the edges of an event-flow graph and those of an integration tree. Edges of an event-flow graph are ordered pairs of the form (e_x, e_y), where e_x and e_y are events, whereas edges of the

integration tree are ordered pairs of the form (C_x, C_y), where C_x and C_y are components. Each edge of the integration tree represents a set of edges with events. An edge (C_x, C_y) represents the set of all edges (e_y, e_z), where e_y is an event in component C_x that invokes C_y. Assume the existence of a function **EventEdges** that takes a set of integration-tree edges and returns its corresponding set of edges in terms of events.

The set of modifications obtained above are used to identify affected test cases. Specifically, the following two sets are used to identify affected test cases:

1. vertices_deleted, and
2. edges_deleted ← efg_edges_deleted ∪ Event-edges (comp_edges_deleted).

As noted earlier, new vertices and edges cannot affect test cases. To aid in the identification of affected test cases, the event-sequence checker uses bit vectors associated with each test case. These bit vectors contain information about the events and edges used by each test case. If a test case uses an event (or edge), then the event's (or edge's) bit is set in the bit vector for that test case. The following bit vectors are associated with each test case T:

- **EVENTS-USED** represent the events used by T. Its length is $|E|$, where E is the set of events in the GUI.
- **EDGES-USED** represent the edges covered by T. Its length is $|D|$, where D is the set of all the edges in the event-flow graphs and integration tree of the GUI.

Examining the above bit vectors for each modification, the event-sequence checker identifies test cases that were affected by each modification. For example, if an event e is deleted from the GUI, then all test cases whose **EVENTS-USED** bit vector's e^{th} bit is set are affected.

5.1.3 Expected State Checker

The expected state checker determines whether the expected state sequence associated with each test case is correct. If the initial state of a test case is reachable and its event sequence is legal, then the test case can technically be executed on the GUI. However, if the

preconditions/effects of an event have been modified in the GUI then the expected state sequence associated with this test case may be incorrect. Such modifications are detected by comparing the preconditions and effects of the original and modified operator for each event.

Test Case Regenerator

In this section, the derivation of planning operators and how AI planning techniques are used to regenerate test cases is described. An algorithm that performs a restricted form of hierarchical planning with caching is presented that employs new hierarchical operators and leads to an improvement in planning efficiency.

5.2.1 Setting up the Planning Problem

Setting up a planning problem requires performing two related activities: (1) defining planning operators in terms of preconditions and effects, and (2) describing tasks in the form of initial and goal states [24]. This section provides details of these two activities in the context of using planning for test case regeneration.

5.2.2 Modeling Planning Operators

For a given GUI, the simplest approach to obtain planning operators would be to identify one operator for each GUI event (Open, File, Cut, Paste, etc.) directly from the GUI representation, ignoring the GUI's component hierarchy. For the remainder of this paper, these operators, presented earlier in Section 4.2, are called primitive operators. When developing the GUI representation, the test designer defines the preconditions and effects for all these operators. Although conceptually simple, this approach is inefficient for regenerating test cases for GUIs as it results in a large number of operators.

An alternative modeling scheme, and the one used in this test case regenerator, uses the component hierarchy and creates high-level operators that are decomposable into sequences of lower level ones. These high-level operators are called system-interaction operators and

component operators. The goal of creating these high-level operators is to control the size of the planning problem by dividing it into several smaller planning problems. Intuitively, the system-interaction operators fold a sequence of menu-open or unrestricted-focus events and a system-interaction event into a single operator, whereas component operators encapsulate the events of the component by treating the interaction within that component as a separate planning problem. Component operators need to be decomposed into low-level plans by an explicit call to the planner. Details of these operators are presented next.

The first type of high-level operators is called system-interaction operators.

> **Definition**: A ***system-interaction operator*** is a single operator that represents a sequence of zero or more menu-open and unrestricted-focus events followed by a system-interaction event.

Consider a small part of a GUI: one pull-down menu with one option (**Edit**) that can be opened to give more options, i.e., **Cut** and **Paste**. The events available to the user are **Edit**, **Cut** and **Paste**. **Edit** is a menu-open event, and **Cut** and **Paste** are system-interaction events. Using this information the following two system-interaction operators are obtained.
EDIT_CUT = <Edit, Cut>
EDIT_PASTE = <Edit, Paste>

The above is an example of an operator-event mapping that relates system-interaction operators to GUI events. The operator-event mappings fold the menu-open and unrestricted focus events into the system interaction operator, thereby reducing the total number of operators made available to the planner, resulting in planning efficiency. These mappings are used to replace the system-interaction operators by their corresponding GUI events when regenerating the final test case.

In the above example, the events Edit, Cut and Paste are hidden from the planner, and only the system-interaction operators, namely, EDIT_CUT and EDIT_PASTE, are made available to the planner. This abstraction prevents regeneration of test cases in which Edit is used in isolation, i.e., the model forces the use of Edit either with Cut or with

Paste, thereby restricting attention to meaningful interactions with the underlying software.
The second type of high-level operators is called component operators.

> **Definition**: A ***component operator*** encapsulates the events of the underlying component by creating a new planning problem and its solution represents the events a user might generate during the focused interaction.

The component operators employ the component hierarchy of the GUI so that test cases can be regenerated for each component, thereby resulting in greater efficiency. For example, consider a small part of the xfig's GUI shown in Figure 15(a): two restricted-focus events, namely Open and SaveAs used to invoke two components called Open and SaveAs respectively. The events in both windows are quite similar. For Open the user can exit after pressing Open or Cancel; for SaveAs the user can exit after pressing Save or Cancel. For simplicity, assume that the complete set of events available is Open, SaveAs, Open.Select, Open.Up, Open.Cancel, Open.Open, SaveAs.Select, SaveAs.Up, SaveAs.Cancel, and SaveAs.Save. (Note that the component name is used to disambiguate events.) Once the user selects Open, the focus is restricted to Open.Select, Open.Up, Open.Cancel, and Open.Open. Similarly, when the user selects SaveAs, the focus is restricted to SaveAs.Select, SaveAs.Up, SaveAs.Cancel, and SaveAs.Save. Two component operators called File_Open and File_SaveAs are obtained.

The component operator is a complex structure since it contains all the necessary elements of a planning problem, including the initial and goal states, the set of objects, and the set of operators. The *prefix* of the component operator is the sequence of menu-open and unrestricted-focus events that lead to the restricted focus event, which invokes the component in question. This sequence of events is stored in the operator-event mappings. For the example of Figure 15(a), the following two operator-event mappings are obtained, one for each component operator:
File_Open = <File, Open>, and
File_SaveAs = <File, SaveAs>.

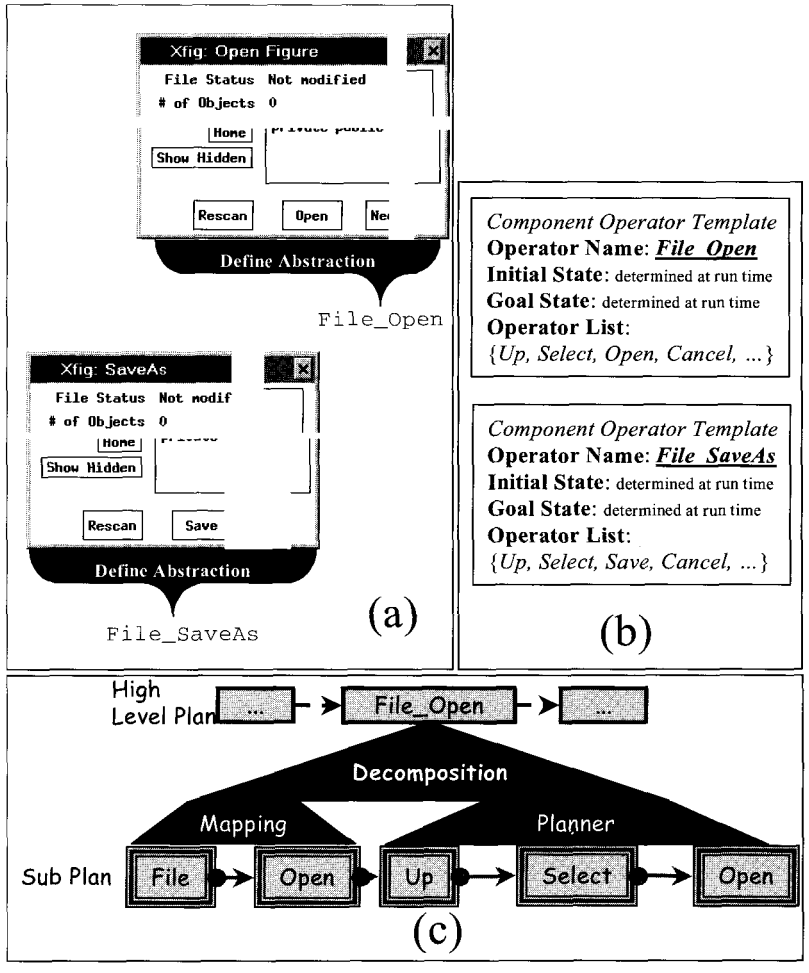


Figure 15. (a) Open and SaveAs Windows as Component Operators, (b) Component Operator Templates, and (c) Decomposition of the Component Operator Using Operator-event Mappings and Making a Separate Call to the Planner to Yield a Sub-plan

The *suffix* of the component operator represents the modal dialog. A component operator definition template is created for each component operator. This template contains all the essential elements of the planning problem, i.e., the set of operators that are available during the interaction

with the component and initial and goal states, both determined dynamically at the point before the call. The component operator definition template created for each operator is shown in Figure 15(b).

The component operator is decomposed in two steps: (1) using the operator-events mappings to obtain the component operator prefix, and (2) explicitly calling the planner to obtain the component operator suffix. Both the prefix and suffix are then substituted back into the high-level plan. At the highest level of abstraction, the planner will use the component operators, i.e., **File_Open** and **File_SaveAs**, to construct plans. For example, in Figure 15(c), the high-level plan contains **File_Open**. Decomposing **File_Open** requires (1) retrieving the corresponding GUI events from the stored operator-event mappings (**File**, **Open**), and (2) invoking the planner, which returns the sub-plan (**Up**, **Select**, **Open**). **File_Open** is then replaced by the sequence (**File**, **Open**, **Up**, **Select**, **Open**). Since the higher-level planning problem has already been solved before invoking the planner for the component operator, the preconditions and effects of the high-level component operator are used to determine the initial and goal states of the sub-plan.

5.2.3 Modeling the Tasks Using Initial and Goal States

Once all the operators have been identified and defined, the next step is to model the tasks using initial and goal states. Recall that GUI states are represented by a set of properties of GUI objects. Figure 16 shows an example of a task for **xfig**. Figure 16(a) shows the initial state: a collection of files stored in a directory hierarchy. The contents of the file **doc2.fig** is shown in an **xfig** window, and the directory structure is shown in an Exploring window. Assume that the initial state contains a description of the directory structure, the location of the files, and the contents of each file. Using these files and **xfig's** GUI, a goal of creating the new figure shown in Figure 16(b) and then storing it in file **new.fig** in the **/root/public** directory is defined. Figure 16(b) shows this goal state that contains, in addition to the old files, a new file stored in **/root/public** directory. Note that **new.fig** can be obtained by loading file **doc2.fig** and inserting text. The code for the initial state and the changes needed to

achieve the goal state is shown in Figure 17. Once the task has been specified, the system automatically regenerates test cases that achieve the goal.

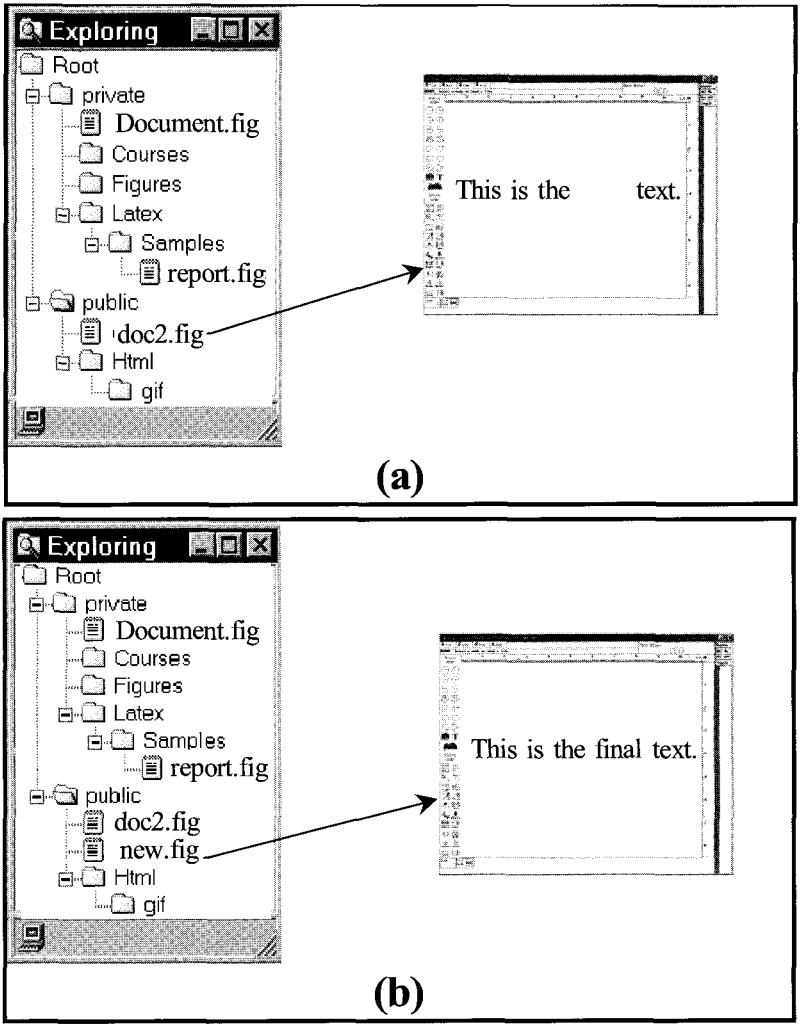


Figure 16. A Task for the Planning System;(a) the Initial State, and (b) the Goal State

Initial State:

```
isCurrent(root)
contains(root private)
contains(private Figures)
contains(private Latex)
contains(Latex Samples)
contains(private Courses)
contains(private Thesis)
contains(root public)
contains(public html)
contains(html gif)
containsfile(gif doc2.fig)
containsfile(private
            Document.fig)
containsfile(Samples
            report.fig)
currentFont(Times Normal
            12pt)
in(doc2.fig This)
in(doc2.fig is)
in(doc2.fig the)
in(doc2.fig text.)
isText(This)
isText(is)
isText(the)
isText(text)
after(This is)
after(is the)
after(the text.)
font(This Times Normal 12pt)
font(is Times Normal 12pt)
font(the Times Normal 12pt)
font(text. Times Normal
            12pt)
.....
```

Similar descriptions for ***Document.fig*** *and* ***report.fig***

Goal State:

```
containsfile(public new.fig)
in(new.fig This)
in(new.fig is)
in(new.fig the)
in(new.fig final)
in(new.fig text.)
after(This is)
after(is the)
after(the final)
after(final text.)
font(This Times Normal 12pt)
font(is Times Normal 12pt)
font(the Times Normal 12pt)
font(final Times Normal
            12pt)
font(text. Times Normal
            12pt)
.....
```

Figure 17. Initial State and the changes needed to reach the Goal State

5.2.4 Generating Plans

The test designer begins the regeneration of affected test cases by inputting the defined operators and then identifying a task, such as the one shown in Figure 16, that is defined in terms of an initial state and a goal state. The planner automatically regenerates a set of test cases that achieve the goal. An example of a plan is shown in Figure 18. (Note that **TypeInText()** is a keyboard event). This plan is a high-level plan that must be translated into primitive GUI events. The translation process makes use of the operator-event mappings stored during the modeling process. One such translation is shown in Figure 19. This figure shows

the component operators contained in the high-level plan are decomposed by (1) inserting the expansion from the operator-event mappings, and (2) making an additional call to the planner.

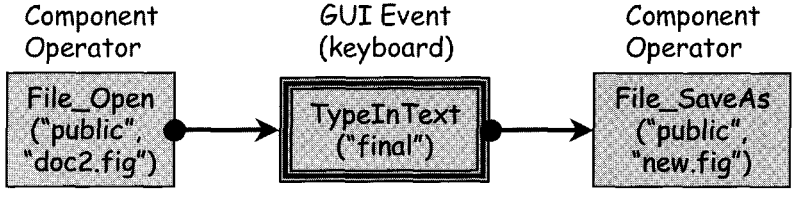


Figure 18. A Plan Consisting of Component Operators and a GUI Event

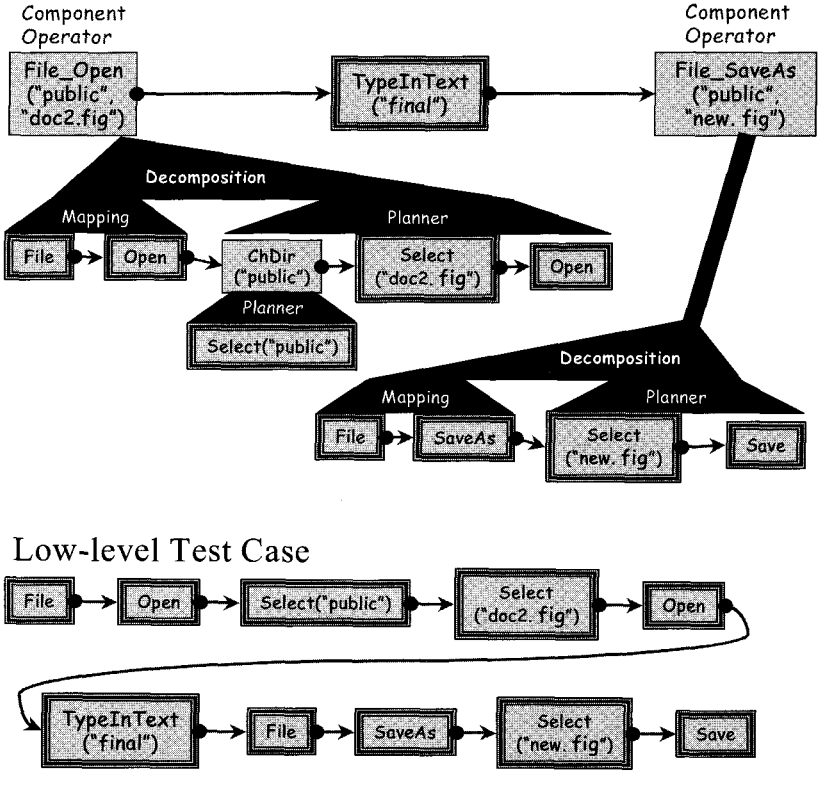


Figure 19. Expanding the Higher Level Plan

5.2.5 Using the Original Test Case as a Cache

Since parts of the original test case are still valid, they can be reused. We cache the parts of the test case so that they can be reused to generate the new test case. Instead of invoking the planner for each subtask, we check if a valid part of the test case already exists. We then reuse that part of the test case.

5.2.6 Algorithm for Regenerating Test Cases

The test case regeneration algorithm is shown in Figure 20. The parameters (lines 1..5) include all the components of a planning problem and a threshold (***T***) that controls the looping in the algorithm. The loop (lines 8..12) contains a lookup to the cache for the relevant subplan. If it exists, then the while-loop condition short-circuits and ***p*** is returned. If there is no matching plan in the cache, then an explicit call is made to the planner (Φ). The returned plan ***p*** is recorded with the operator set, so that the planner can return an alternative plan in the next iteration (line 11). At the end of this loop, **planList** contains all the partial-order plans. Each partial-order plan is then linearized (lines 13..16), leading to multiple linear plans. Initially the test cases are high-level linear plans (line 17). The decomposition process leads to lower level test cases. The high-level operators in the plan need to be expanded/decomposed to get lower level test cases. If the step is a system-interaction operator, then the operator-event mappings are used to expand it (lines 20..22). However, if the step is a component operator, then it is decomposed to a lower level test case by (1) obtaining the GUI events from the operator-event mappings, (2) calling the planner to obtain the sub-plan, and (3) substituting both these results into the higher level plan. Extraction functions are used to access the planning problem's components (lines 24..27). The lowest level test cases, consisting of GUI events, are returned as a result of the algorithm (line 33).

```
                                                                          Lines
Algorithm :: GenTestCases(
  Λ = Operator Set;                                                         1
  D = Set of Objects;                                                       2
  I = Initial State;                                                        3
  G = Goal State;                                                           4
  T = Threshold) {                                                          5

  planList ← {};                                                            6
  c ← 0;                                                                    7
  /* Successive calls to the planner (Φ),
  modifying the operators before each call */
  WHILE (((p == CACHE(Λ, D, I, G)) || ((p == Φ(Λ, D, I, G)) ! = NO_PLAN))    8
    && (c < T) DO {                                                         9
   InsertInList(p, planList);                                              10
   Λ ← RecordPlan(Λ, p);                                                   11
   c + +}                                                                  12

  linearPlans ← {};/* No linear Plans yet */                               13
  /* Linearize all partial order plans */
  FORALL e ∈ planList DO {                                                 14
   L ← Linearize(e);                                                       15
   InsertInList(L, linearPlans)}                                           16

  testCases ← linearPlans;                                                 17
  /* decomposing the testCases */
  FORALL tc ∈ testCases DO {                                               18
   FORALL C ∈ Steps(tc) DO {                                               19
    IF (C == systemInteractionOperator) THEN {                             20
     newC ← lookup(Mappings, C);                                           21
     REPLACE C WITH newC IN tc}                                            22
    ELSEIF (C == componentOperator) THEN {                                 23
     AC ← OperatorSet(C);                                                  24
     GC ← Goal(C);                                                         25
     IC ← Initial(C);                                                      26
     DC ← ObjectSet(C);                                                    27
     /* Generate the lower level test cases */
     newC ← APPEND(lookup(Mappings, C),
      GenTestCases(AC, DC, IC, GC, T));                                    28

     FORALL nc ∈ newC DO {                                                 29
      copyOftc ← tc;                                                       30
      REPLACE C WITH nc IN copyOftc;                                       31
      APPEND copyOftc TO testCases}}}}                                     32
  RETURN(testCases)}                                                       33
```

Figure 20. The Complete Algorithm for Regenerating Test Cases

Expected State Regenerator

We now explain how we use the model of the GUI to regenerate the expected state of a GUI test case. Recall that events are modeled as state transducers. For any state S_0, and event sequence e_1; e_2; ...; e_n, the sequence of states S_1; S_2; ...; S_n such that $S_i = e_i(S_{i-1})$ for $i = 1, ..., n$ represents the expected state of the GUI after each action is executed, starting in S_0. The next state is obtained from the current state S_c and the operator's effects e as follows:

1. Delete all literals in S_c that unify with a negated literal in e, and
2. add all positive literals in e.

Thus, using a formal model of a GUI, we can derive the expected state, given an initial state and a sequence of events. This technique is shown in the algorithm in Figure 21.

```
PROCEDURE: ExpStateGen(                                                    1
 currentState, /* properties, {p1,p2,p3,...,pn} - the State of the GUI*/   2
 event, /* step of the test case - eventName(parameters)*/                 3
 operators /* {Op1,Op2,Op3,...,Opn}. */) {                                4
  opDef ⟵ Lookup(event, operators); /* get operator for event */          5
  op ⟵ Bind(opDef, event); /* bind all variables in op def. */           6
  p ⟵ preconditions(op); /* extract the preconditions of the operator */ 7
  IF(Satisfied(p, currentState) == FAILED)                                 8
    RETURN(INVALID_STATE);                                                 9
  eff ⟵ effects(op); /*extract the effects of the operator*/              10
  newState ⟵ currentState;                                                11
  FOREACH (f ∈ eff) {/*delete all properties that are negated in effects*/ 12
    IF (negated(f)) delete f from newState;                               13
  FOREACH (f ∈ eff) {/*insert all properties that are positive in effects*/ 14
    IF (positive(f)) insert f in newState;                                15
  RETURN(newState);}                                                      16
```

Figure 21. The Algorithm for Generating the Expected State

The procedure **ExpStateGen** takes as input the current state of the GUI (**currentState**), the event to be executed on the GUI, and the GUI (operators). Every event in the test case has a corresponding operator

definition (line 5). The event contains the actual parameters of the operator definition, which are substituted for the formal parameters (line 6). **ExpStateGen** performs an extra check to determine if the preconditions of the operator are satisfied in the current state (lines 8..9). If they are not satisfied, then the expected state is an error state. If the preconditions are satisfied, the new state is computed by applying the effects of the operator. If the effects contain a negated property, then it is deleted from the new state (lines 12..13) and if it contains a positive property, it is inserted (lines 14..15) in the new state. The result **newState** is returned to the calling algorithm, which inserts it in the test case.

6. Experiments

To explore the practicality of our new regression test case selection and replanning techniques, we implemented the regression tester and evaluated its performance on **Xfig 3.2 patchlevel 0-beta4 (Protocol 3.2)** and **Xfig 3.2 patchlevel 3c (Protocol 3.2)**. We used the former as our *original* GUI and the latter as our *modified* GUI. We specifically wanted to determine (1) how many of the original test cases are affected by the modifications, (2) how much time it takes to selectively generate the regression test suite and how it compares to regenerating all test cases, (3) how much total time is saved in terms of regeneration and re-execution, (4) what affect cache has on the regeneration process. The study was conducted on a 1.7 GHz Pentium Workstation with 1 GB of RAM. All execution times are CPU time. The study consisted of the following steps:

1. *Generating test cases for the original software*: There are several automated test case generation techniques that we may have used: (1) task-based using AI planning [24], (2) structural using event-flow graphs and integration trees [25], (3) user-model based using genetic algorithms [14], and (4) random. Since we were going to use the planning-based regression testing technique, we chose to use planning to generate the test cases in the first place. We used the Unix-based IPP [15, 16] planner. We identified 100 tasks for which

we generated 1000 test cases. Note that we can generate multiple test cases for a given task. The test cases varied from length 6 to 124. The total time taken to generate the test cases was 1239 seconds.

2. *Implementation*: All the components of the regression tester were implemented, except the planner that we used off-the-shelf. Specifically, the test case selector was implemented in Perl and the expected-state regenerator was implemented in C.
3. *Identifying affected test cases*: The test case selector determined the number of affected test cases by comparing the event-flow graphs of the original and modified software. The results of this step are summarized in Figure 22. Note the initial state was never made unreachable since we used a **LaunchApplication** operator: the initial state of each test case consisted of the files only, which did not change when the software was modified. The graph also shows the reason why the test cases were affected. Note that the total exceeds the total number of affected test cases since a test case could be affected both because its event sequence and expected state was invalid/illegal. The remaining test cases (unaffected) need not be rerun on the modified software since they are not modification traversing. As shown in Figure 22, 429 (42.9%) test cases are unaffected. Figure 23 shows the time taken by the test case selector. In all, the test case selector identified 571 affected test cases in 391 seconds.
4. *Regenerating test cases*: The test case regenerator successfully regenerated the affected test cases using the tasks. In all, 543 test cases were regenerated in 212 seconds.
5. *Regenerating expected state*: The expected state regenerator successfully regenerated the affected test cases' expected state. Note that we had to regenerate expected states for only 192 (543-351) test cases since we regenerated the 351 test cases using the planner that also produces the expected state. In all, expected states for 192 test cases were regenerated in 91 seconds. The total regression test generation time was 694 seconds. The overall regression testing time is reduced to 56% of the original, resulting in considerable overall savings in test case execution time (Figure 24).

6. *Cache vs. no Cache*: We wanted to see what affect the cache had on the performance of the test case regenerator. So, we also generated test cases without using the cache. The results are shown in Figure 25. The x-axis shows the test case length, the y-axis shows the average time taken to generate the test cases. In general, the cache cut the regression test case generation time by half.

The regression test cases obtained above form an important part of the regression testing test suite. The test designer will also need to generate additional test cases to check the new parts of the GUI.

In this experiment we have successfully demonstrated that our technique is practical and can be used to select test cases for GUI regression testing. The use of this technique can help reduce the cost of GUI regression testing. Our experience with GUI testing has shown that the currently employed techniques, which are largely manual aided with capture/replay tools, require weeks to develop only a few hundred test cases.

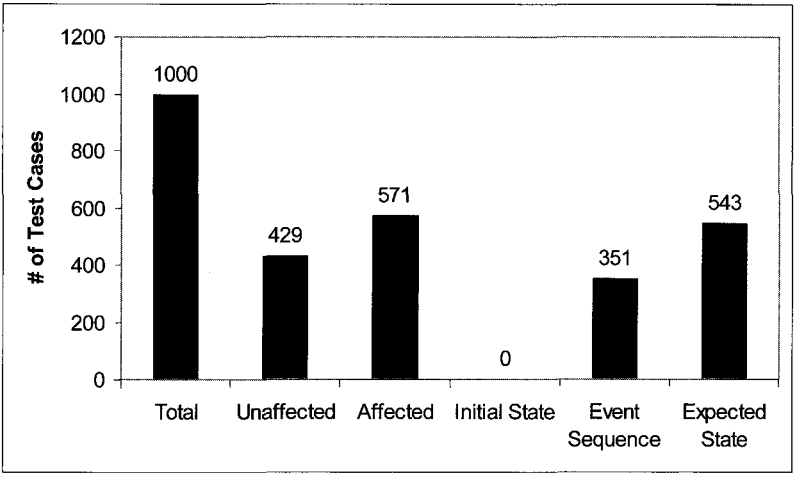


Figure 22. Identifying Affected Test Cases

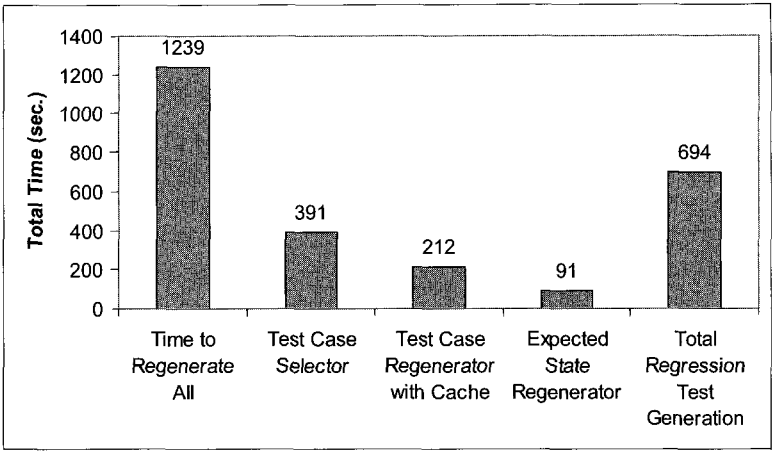


Figure 23. Time Taken to Perform Regression Test Selection/Regeneration vs. Time Taken to Regenerate All

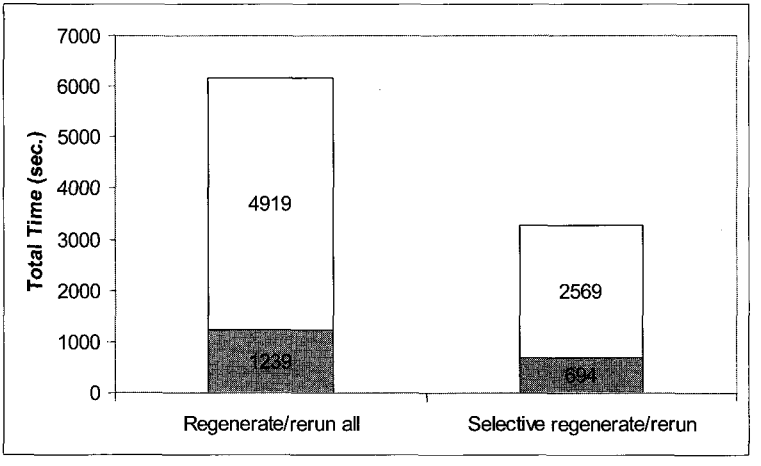


Figure 24. The Total Regression Testing Time vs.Regenerating/Rerunning All

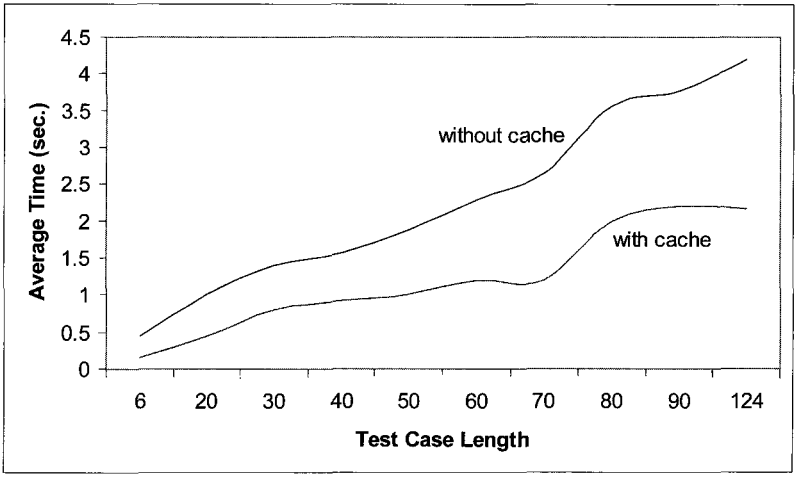


Figure 25. Replanning Using a Cache vs.Not Using a Cache

7. Conclusions

This paper presented a new technique for regression testing based on replanning affected GUI test cases by associating a task with each test case. Affected test cases are identified by employing the specifications of the GUI. Differences between the event-flow graphs and integration trees of the original and modified GUIs are obtained to identify affected test cases. Results of a case study performed on Xfig 3.2 patchlevel 0-beta4 (Protocol 3.2) and Xfig 3.2 patchlevel 3c (Protocol 3.2) show that the regression testing technique is efficient, in that it identifies a large percentage of test cases that need not be rerun on the modified GUI and helps select a set of affected test cases that are efficiently regenerated and rerun.

The modifications discussed in this paper were complex event-level and component-level modifications. Other low-level modifications may also be made to a GUI. For example, new keyboard shortcuts may be introduced in the modified GUI or the physical locations of buttons/menus may be changed. Such changes do not affect the test cases since all the events in the test case are represented by logical symbols

rather than low-level physical locations on the screen or keyboard shortcuts used to generate them. A mapping between logical events and the corresponding physical actions used to generate them is maintained. At test case execution time, this mapping is used to generate physical actions for each logical event. When these events/shortcuts are changed from one GUI version to the next, the mappings are modified without affecting the test cases.

Modification of conventional software also affects test cases (also known as obsolete test cases [34]) that are simply discarded. Studies need to be conducted to determine whether the task-based technique developed in this paper can be extended for conventional software.

References

[1] Agrawal, H., Horgan, J. R., Krauser, E. W., and London, S. A. "Incremental regression testing". In *Proceedings of the Conference on Software Maintenance* (Washington, Sep. 1993), pp. 348 – 357.

[2] Ball, T. "On the limit of control flow analysis for regression test selection". In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis* (ISSTA-98) (New York, Mar.2 –5 1998), vol. 23, 2 of ACM Software Engineering Notes, pp. 134 – 142.

[3] Beizer, B. *Software Testing Techniques*, 2nd ed. Van Nostrand Reinhold, New York, 1990.

[4] Benedusi, P., Cimitile, A., and DeCarlini, U. "Post-maintenance testing based on path change analysis". In *Proceedings of the IEEE Conference on Software Maintenance* (1988), pp. 352 – 368.

[5] Binkley, D. "Reducing the cost of regression testing by semantics guided test case selection". In *Proceedings of the International Conference on Software Maintenance* (Washington, Oct. 17 – 20 1995), G.Caldiera and K.Bennett, Eds., IEEE Computer Society Press, pp. 251 – 263.

[6] Binkley, D. "Semantics guided regression test cost reduction". *IEEE Transactions on Software Engineering*, 23, 8 (Aug.1997), 498 – 516.

[7] Frank, M., de, J. J. G., Gieskens, D., and Foley, J. D. "Building user interfaces interactively using re-and postconditions". In *Proceedings of CHI '92* (1992).

[8] Green, M. *The Design of Graphical User Interfaces*. Ph.d. thesis, Department of Computer Science, University of Toronto, 1985.

[9] Harrold, M. J., Gupta, R., and Soffa, M. L. "A methodology for controlling the size of a test suite". *ACM Transactions of Software Engineering and Methodology* 2, 3 (July 1993), 270 – 285.

[10] Harrold, M. J., McGregor, J. D., and Fitzpatrick, K. J. "Incremental testing of object-oriented class structures". In *Proceedings: 14th International Conference on Software Engineering* (1992), pp. 68 – 80.

[11] Harrold, M. J., and Soffa, M. L. "Interprocedual data flow testing". In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Testing, Analysis, and Verification* (TAV3) (1989), pp. 158 – 167.

[12] Jonsson, A. K., and Ginsberg, M. L. "Procedural reasoning in constraint satisfaction". In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning* (San Francisco, Nov. 5 – 8 1996), L. C. Aiello, J. Doyle, and S. Shapiro, Eds., Morgan Kaufmann, pp. 160 – 173.

[13] Kambhampati, S. "Can we bridge refinement-based and SAT-based planning techniques?" In *Proceedings of the 15th International Joint Conference on Artificial Intelligence* (IJCAI-97) (1997), M. Pollack, Ed., Morgan Kaufmann, pp. 44 –49.

[14] Kasik, D. J., and George, H. G. "Toward automatic generation of novice user test scripts". In *Proceedings of the Conference on Human Factors in Computing Systems: Common Ground* (New York, 13 –18 Apr. 1996), ACM Press, pp. 244 – 251.

[15] Koehler, J., Nebel, B., Hoffman, J., and Dimopoulos, Y. *Extending planning graphs to an ADL subset.* Lecture Notes in Computer Science 1348 (1997), 273.

[16] Koehler, J., Nebel, B., Hoffman, J., and Dimopoulos, Y. "Extending planning graphs to an ADL subset". In *Proceedings of the 4th European Conference on Planning (ECP-97): Recent Advances in AI Planning* (Berlin, Sept. 24 –26 1997), S. Steel and R. Alami, Eds., vol. 1348 of LNAI, Springer, pp. 273 – 285.

[17] Kung, D. C., Gao, J., Hsia, P., Toyoshima, Y., and Chen, C. "On regression testing of object-oriented programs". *The Journal of Systems and Software 32, 1* (Jan. 1996), 21 – 31.

[18] Lifschitz, V. "On the semantics of STRIPS. "In Reasoning about Actions and Plans:" *Proceedings of the 1986 Workshop* (Timberline, Oregon, June-July 1986), M. P. George and A. L. Lansky, Eds., Morgan Kaufmann, pp.1 – 9.

[19] Mahajan, R., and Shneiderman, B. "Visual & textual consistency checking tools for graphical user interfaces". *Technical Report CS-TR-3639*, University of Maryland, College Park, May 1996.

[20] McCarthy, J. *Situations, actions, and causal laws*. Memo 2, Stanford University Artificial Intelligence Project, Stanford, California, 1963.

[21] Memon, A. M. *A Comprehensive Framework for Testing Graphical User Interfaces*. Ph.D. thesis, Department of Computer Science, University of Pittsburgh, July 2001.

[22] Memon, A. M., Pollack, M. E., and Soffa, M. L. "Using a goal-driven approach to generate test cases for GUIs". In *Proceedings of the 21st International Conference on Software Engineering* (May 1999), ACM Press, pp. 257 – 266.

[23] Memon, A. M., Pollack, M. E., and Soffa, M. L. "Automated test oracles for GUIs". In *Proceedings of the ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering* (FSE-8) (NY, Nov. 8 – 10 2000), pp. 30 – 39.

[24] Memon, A. M., Pollack, M. E., and Soffa, M. L. "Hierarchical GUI test case generation using automated planning". *IEEE Transactions on Software Engineering* 27, 2(Feb. 2001), 144 – 155.

[25] Memon, A. M., Soffa, M. L., and Pollack, M. E. "Coverage criteria for GUI testing". In *Proceedings of the 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)* (Sept. 2001), pp. 256 – 267.

[26] Myers, B. A. "User interface software tools". *ACM Transactions on Computer-Human Interaction 2, 1* (1995), 64 – 103.

[27] Pednault, E.P.D. "ADL: Exploring the middle ground between STRIPS and the situation calculus". In *Proceedings of KR '89* (Toronto, Canada, pp. 324 – 331, May 1989).

[28] Pollock, L., and Soffa, M. L. "Incremental global reoptimization of programs". *ACM Transactions on Programming Languages and Systems* 14, 2 (Apr. 1992), 173 – 200.

[29] Pressman, R. S. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 1994.

[30] Rosenblum, D., and Rothermel, G. "A comparative study of regression test selection techniques". In *Proceedings of the IEEE Computer Society 2nd International Workshop on Empirical Studies of Software maintenance* (Oct. 1997), pp.89 – 94.

[31] Rosenblum, D. S., and Weyuker, E. J. "Predicting the cost-effectiveness of regression testing strategies". In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering* (New York, Oct. 16 –18 1996), vol. 21 of ACM Software Engineering Notes, ACM Press, pp. 118 –126.

[32] Rosenblum, D. S., and Weyuker, E. J. "Using coverage information to predict the cost-effectiveness of regression testing strategies". *IEEE Transactions on Software Engineering* 23, 3 (Mar. 1997), 146 – 156.

[33] Rothermel, G., and Harrold, M. J. "A safe, efficient algorithm for regression test selection". In *Proceedings of the Conference on Software Maintenance* (1993), pp. 358 – 369.

[34] Rothermel, G., and Harrold, M. J. "A safe, efficient regression test selection technique". *ACM Transactions on Software Engineering and Methodology*, 2 (Apr. 1997), 173 – 210.

[35] Rothermel, G., and Harrold, M. J. "Empirical studies of a safe regression test selection technique". *IEEE Transactions on Software Engineering* 24, 6 (June 1998), 401 – 419.

[36] Rothermel, G., Harrold, M. J., Ostrin, J., and Hong, C. "An empirical study of the effects of minimization on the fault detection capabilities of test suites". In *Proceedings; International Conference on Software Maintenance* (1998), pp. 34 – 43.

[37] Schach, S. R. *Software Engineering*, s[nd] ed. Richard D. Irwin/Aksen Associates, 1993.

[38] Weld, D. S. "An introduction to least commitment planning". *AI Magazine*, 15, 4 (1994), 27 – 61.

[39] Weld, D. S. "Recent advances in AI planning". *AI Magazine 20*, 1 (Spring 1999), 55 – 64.

[40] White, L. "Regression testing of GUI event interactions". In *Proceedings of the International Conference on Software Maintenance* (Washington, Nov. 4 – 8 1996), pp. 350 – 35.

This page is intentionally left blank

CHAPTER 4

TEST SET GENERATION AND REDUCTION WITH ARTIFICIAL NEURAL NETWORKS

Prachi Saraph, Abraham Kandel

University of South Florida
Tampa FL: 33620, USA
E-mail: {saraph, kandel}@csee.usf.edu

Mark Last

Department of Information Systems Engineering
Ben-Gurion University of the Negev
Beer-Sheva 84105 ,Israel
E-mail: mlast@bgumail.bgu.ac.il

Reducing the number of test cases results directly in the saving of software testing resources. Based on the success of Neural Networks as classifiers in many fields we propose to use neural networks for automated input-output analysis of data-driven programs. Identifying input-output relationships, ranking input features and building equivalence classes of input attributes for a given code are three important outcomes of this research in addition to reducing the number of test cases. The proposed methodology is based on the three-phase algorithm for efficient network pruning developed by R. Setiono and his colleagues. A detailed study shows that the neural network pruning and rule-extraction can significantly reduce the number of test cases.

1. Introduction

The primary purpose of the software testing process is to create reliable, safe and high quality software that would conform to the requirements specification. Two common approaches to software testing are black-box

testing and white-box testing. White-box testing uses the source code of the software and black box (function) testing analyses the software inputs and outputs rather than examining the actual code.

For a program or a system having a large number of inputs, the number of corresponding black-box test cases involving all possible combinations of inputs is huge. Executing all the test cases requires tremendous resources. Hence the number of test cases to be executed needs to be limited as per the availability of resources. A method or criterion needs to be established to choose the most important and significant test cases that are sufficient to reveal the majority of software faults. Input-output (I-O) analysis of the program identifies the input attributes which affect the value of a particular output the most. In this paper, we show that I-O analysis performed using artificial neural networks (NNs) can reduce number of test cases significantly. It can identify important attributes and also rank them.

The organization of this paper is as follows: Section 2 explains the basics of reducing the number of test cases and covers existing approaches to finding I-O relationships between software attributes. Section 3 illustrates the role of ANNs in the software testing process. Section 4 explains the proposed methodology. In Section 5, we describe the design of experiments and the software program used as a case study for test case reduction. Section 6 concludes the paper with presentation and analysis of the experimental results. The same section outlines the future scope of this research.

2. Software Testing Methods

Errors occur in the software at various stages of the Software Development Life Cycle (SDLC), which include requirements specifications, system design and coding. According to Paul Jorgenson [1] the two main reasons for testing are:

- To make a judgement about quality or acceptability of the software
- To identify and fix problems (faults) in the software

It is a well-known fact that discovering and fixing software faults before the release of the software reduces the maintenance cost significantly.

Unit, integration, and system testing are the main steps in software testing. Black-box testing and white-box testing are two approaches to testing. Black-box testing views the software system as a mapping from inputs to outputs and ignores the processing being done inside completely. The process of black-box testing entails establishing the necessary preconditions, providing the test case inputs, observing the actual outputs and comparing those with the expected outputs to determine whether the test passed. Equivalence class testing and boundary value analysis are two types of black-box testing. Both these approaches are covered in depth in [1] and [2]. In equivalence class testing the data values taken by an input variable are partitioned into ranges (equivalence classes). At least one value from each equivalence class is tested. Test cases contain the combinations of values taken from all equivalence classes of every input. Boundary value analysis is very important for testing any software as most of the errors are caused at the boundary values of the inputs. Test cases for boundary value analysis contain values at the boundary of the possible range for every input. White-box approach works with the source code directly. Hence the white-box test cases are based on how the specification has actually been implemented. The focus is on what is being tested, what part of the code is being executed by running a certain set of test cases. Structural methods like basis-path testing and define/use testing which use program graphs and decision path graphs require precise measurements and mathematical analysis [1], [2].

The main objective of test planning is to determine a set of test cases for a given software system. A test case is defined by a set of input values and a set of expected output values. A good test case according to [2]:

- Should have a high probability of catching an error
- Should not be redundant
- Should be the best of its breed
- Should be neither too complex nor too simple

For a software system having many input and output attributes, the number of combinatorial test cases is so high that it is impossible to list and execute all of them [3]. At the same time it is necessary to make sure that nearly all the software faults have been detected and fixed. Consequently, the choice of test cases is a non-trivial problem. One solution is to establish the criteria for choosing the most effective, important test cases and removing the redundant, unnecessary ones. There could be numerous perspectives of *effective* or *important* test cases. The number of test cases can be optimized using different methods in both white-box and black-box testing.

The ways of providing automatic testing of the parallelized protocol implementations are reviewed in [3]. It suggests a white-box method for reducing test cases by identifying redundant delay points in the tested code. It builds the flow graph of the software and performs dataflow analysis on the flow graph. Then it identifies the unreachable nodes of this graph as the redundant delay points which are ignored while developing the test cases.

The basic idea of I-O analysis is to reduce the number of combinatorial tests by focusing on those input combinations that affect system output the most [4]. Manual and semi-automatic approaches to determining I-O relationships include structural analysis and execution-oriented analysis [4]. Structural analysis uses program source code whereas execution-oriented analysis just observes program behavior. Two commonly followed techniques for structural analysis are viz. static techniques and dynamic techniques. Static techniques are based on the analysis of source code and dynamic techniques are based on analysis of information recorded during execution. There are merits and demerits of these two approaches. Static techniques determine absence of relationships and may identify non-existing relationships. Dynamic and execution-oriented techniques determine presence of relationships and may not identify all the relationships.

The information about the relationships between input and output attributes of the software can be used for reducing number of test cases [4]. The input-output analysis presented in [4] enables running test cases within resource limitations. However, the methodology can only be

implemented by a human expert, such as an experienced and expensive tester, and it involves a lot of subjective decisions. For example, a tester can analyze system specifications, perform structural analysis of the source code, and observe the results of system execution.

In [5] Last and Kandel present a novel approach of using a data mining algorithm termed IFN (Info-Fuzzy-Networks) [6] to identify I-O relationships automatically. The info-fuzzy method of input-output analysis produces a ranked list of inputs relevant to each output as a result of mining the execution data. The method does not require the knowledge of either the tested code or the requirements, except for a list of system inputs and outputs along with their respective data types. The superiority of IFN for the feature selection task has been shown in [23]. This paper extends the ideas of [5] by replacing info-fuzzy networks with neural networks as data mining and attribute selection tool.

3. Neural Networks and Software Testing

As given by Kohonen [7], NNs can be defined as "massively parallel interconnected networks of simple (usually adaptive) elements and their hierarchical organizations which are intended to interact with objects of the real world in the same way as biological nervous systems do".

3.1. Main Characteristics of Neural Networks

NNs are multilayer perceptrons. The different layers include an input layer (a collection of sensory or non-computational units) one or more hidden layers (a collection of computational units) and an output layer (again a collection of computational units). All these units are connected to each other with links having synaptic weights. An input signal propagates through these layers in the forward direction till it hits the output layer. The most common method for inducing connection weights from training data is the back propagation algorithm.

Architecturally, NNs are inspired by biological structure of a neuron as well as mathematical methods of learning. They have been designed to resemble the human brain in terms of information processing. They have

an input, an activation function, an output and a synaptic weight associated with them. These weights are updated as a part of the training process and reflect the information which is learned during training.

Three reasons for suitability of NNs to the learning task have been elaborated in [8]. These are:

- Straightforward and direct manner of acquiring knowledge and information through the training phase
- Knowledge stored in the form of weights
- Robustness to noise in the input data

A fact that NNs do not have a provision for user explanation capability has also been discussed in [8]. As mentioned in [9] the NNs do not assume any underlying model and learn the concept during the training phase strictly from the training data. There are certain issues which need to be handled before using the NNs as classifiers.

- Inputs and outputs of the network need to be normalized in a numerical form
- No standard method for defining the architecture of an ideal network i.e. number of hidden units and hidden layers exists
- Training NNs using back-propagation is very slow and can converge to a local minima
- Learning process relies heavily on the training data, hence the data presented to NNs has to be a good representative of its sample

The learning power of NNs is attributed to the hidden layer neurons, the synaptic weights and the training algorithm. The advantages of NNs can be summarized as follows:

- Distributed knowledge in the form of weights
- Learning and adapting through efficient knowledge acquisition
- Domain free modeling
- Robustness to noise

Their shortcomings can be listed as:

- Lack of explanation capability
- Time consuming network construction and training
- Hard to determine the optimal number of hidden layers and hidden units
- Convergence not guaranteed in learning algorithms

3. 2. Neural Network Post-Processing

One important aspect of NN training is the predictive accuracy of the network on unseen data. Finding the smallest model that will fit the data and will generalize well on the unseen data is difficult. There are two common approaches to this problem. The first one is training a network that is larger than necessary and then pruning it. The second method is adding nodes and links to the network only when needed.

A review of several pruning algorithms appears in [14]. These algorithms prune the network by removing unnecessary links from it. According to [14] the pruning algorithms can be divided into two groups. In the first group, the criterion used for removing the links is sensitivity of the error function to the removal of the link. The links which affect the error function the least can be removed. On the other hand, the second approach adds a penalty term to the objective function. This term penalizes unnecessary weights by driving them to zero and rewards the important weights. The weights with smaller magnitudes are not likely to affect the accuracy of the network and can be removed. The weights with greater magnitudes need to be retained. A penalty function approach to pruning the NN developed by Setiono is explained in[15] [10] and [16]. The penalty function added to the objective function (the cross entropy function) identifies relevant connections while the weight-decay back-propagation network is being trained. This algorithm gets its name from the fact that it accelerates decay of unimportant weights. A connection weight pruning algorithm which measures sensitivity of a weight in presence of all other weights is employed in [17]. Pruning might be used as a method of feature selection if it is able to remove connections irrelevant to the concept being learned or problem being modeled [14]. This will be demonstrated in the sections to follow.

The predictive accuracy of the NNs tends to be higher than the other learning methods but it is often difficult to understand the induced concept because of their complex architecture. It is therefore desirable to extract classification rules from the network structure to explain the decisions they make. Many rule-extraction algorithms have been reviewed in [8]. The search based methods for rule-extraction often pose constraints on values which can be taken by the hidden units. They

attempt to search the subsets of combinations of inputs that exceed the bias on the hidden unit. Hence they try to restrict the values which could be taken by the hidden unit to reduce the time complexity for the search. Some algorithms do not restrict the hidden unit activation values; one of them is illustrated in [16]. This algorithm discretizes and clusters the hidden unit activation values, links these values with inputs and outputs. Another hidden unit activation value clustering algorithm is presented in [10]. A knowledge based approach that is derived from the function that NN learns rather than the weights in the NN has been explained in [18]. It based on the concept of monotonic regions and sensitivity patterns. A novel approach towards extracting rules which is quite different from many existing search based algorithms is given in [19].

3.3. Neural Networks in Software Testing and Other Domains

NNs have been used in the software testing process before. NN-based mechanism that is able to learn based on past history which test cases are likely to detect faults is presented in [11]. An application of NNs in software testing has been discussed in [12]. It uses a two-layer NN to detect faults within mutated code of a small credit card approval application. In [13], this approach is successfully applied to three software applications. According to [10] and [12] it is advisable to use NNs in regression testing as an automated oracle for evaluating the correctness of subsequent versions of the software system for various reasons, the most significant of which is their generalization capability.

NNs are applied to solve complex problems in a variety of other fields. Despite the fact that the popularly used back-propagation algorithm does not guarantee convergence to global minima, NNs are employed in many real-world applications. Some common fields are pattern recognition, machine learning and data mining.

The use of NNs as a tool for predicting chemically-induced carcinogenesis in rodents by training on data derived from animal tests is explained in [7]. The tasks performed are: training the network, pruning the weight matrix and extracting rules from it. An application of NNs to breast cancer diagnosis has been discussed in [10]. The approach there is

a three phase training algorithm (construction, pruning and rule-extraction) which we will be using in our test reduction methodology. The details of this algorithm will be discussed in Section 4.

4. The NN-based methodology for test case generation and reduction

This section explains the ideas and algorithms which have been used in our approach. We first show how test cases can be reduced with I-O analysis using the approach presented in [5, 16]. We also illustrate this approach with an example. Then we present our methodology for I-O analysis using NNs. We proceed with describing the NN training, pruning and rule-extraction algorithms based on [15], [10], [16] and [20]. Finally, we present the feature ranking method which is based on the pruning algorithm from [15].

4.1. Input-Output Analysis with Artificial Neural Networks

As demonstrated in [4] determining I-O relationships can reduce the number of test cases in functional testing. We have mentioned above that pruning a NN removes unnecessary connections while retaining the significant ones and hence can be used for deducing I-O relationships. Our test case generation and reduction methodology is based on a set of algorithms explained in [20]. A weight decay back-propagation network is built first. The weight decay mechanism assigns bigger magnitudes to important weights and smaller values to the unimportant weights. In the pruning phase these smaller weights are removed without affecting the predictive accuracy of the network. The inputs of the program are then ranked in the order of their importance by observing the order in which the links in the network are removed in the pruning phase. This ranking of features helps identify attributes contributing to the value of output. We can thus focus on the most significant attributes implied by the ranking and use that information for building the test cases. Test cases involving the attributes with higher ranking can be included and test cases involving attributes receiving lower ranks can be eliminated. The

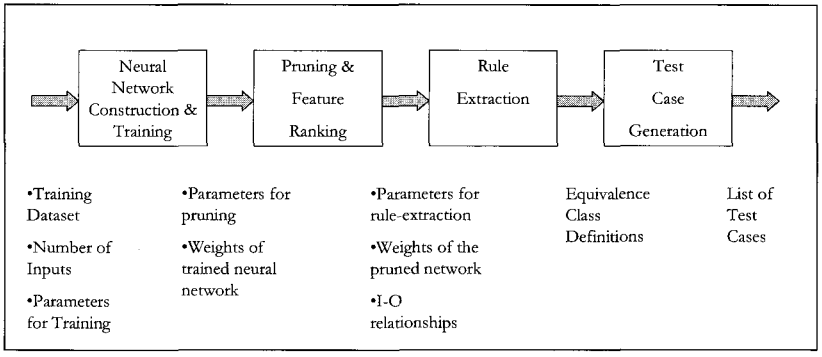


Figure 1. Overview of test generation methodology

rule-extraction phase deduces rules refining the I-O relationships extracted by the pruning phase. It helps building equivalence classes for continuous input attributes. The test cases are built by taking combinations of data values of the inputs. Reduction of size of the input domain results in reduction of test cases.

The tasks involved in our test case generation methodology are as shown in Figure 1. It consists of four phases: the network construction and training, the pruning-feature ranking, rule-extraction and test case generation. After the trained network has been pruned it is easy to associate the links left in the network from input layer to hidden layer with the corresponding input nodes. During the pruning phase we can note the order of removal of the links and associate that order with the input nodes. The result of the pruning phase is the I-O relationships. The rule-extraction phase further defines these relationships in the form of rules. The most important contribution of rule-extraction phase is the equivalence classes built on the data values of continuous inputs. This can further reduce the number of test cases. In the sub-sections below we explain the four phases in detail.

4.2. Network Construction and Training

The steps in the network construction phase include:

- Build a multi-layer NN which meets the specifications of the original software such as number of inputs, number of outputs, types of inputs and their range
- Generate training and test sets of execution data or use available datasets
- Pre-process and normalize the data before presenting it to the NN
- Initialize various parameters of the training process such as learning rate, number of epochs etc.
- Train the NN so that it can map the inputs to the outputs almost the way the original software does

The algorithm used for training the NN is the weight decay back-propagation algorithm. As summarized in [15] penalty-term methods add a penalty function to the objective function of the back-propagation which helps unimportant weights take smaller values during the training phase itself. Now we explain the penalty function approach to pruning feed-forward NN as described in [10], [15], and [20].

Consider a three layer feed-forward NN with the links from every input unit to every hidden unit and from every hidden unit to the output unit. The weights of the connections between the input and hidden layer are denoted by $w_l^j, l = 1,2,3,...,n, j = 1,2,3,...,h$ .n is number of input units and h is number of hidden units. Let v_p^j denote weights from hidden layer to the output layer, $p = 1,2,3,...,o$, where o is the number of output units. Let C_1, C_2, C_3, $\ldots C_o$ be the number of classes for the output. Let there be k input patterns having dimensionality equal to n (number of inputs). Let vector t_i denote the target output for pattern i.

The back-propagation algorithm tries to compute the best set of weights for the given set of examples by minimizing the cross entropy function given as follows:

$$F(w,v) = -\sum_{i=1}^{k}\sum_{p=1}^{o}(t^i{}_p \log S^i{}_p + (1-t^i{}_p)\log(1-S^i{}_p)) \qquad (1)$$

Equation (2) gives the formula used for computing S_p^i which is the actual output of the network, $\sigma(.)$ is the sigmoid function and $\delta(.)$ is the hyperbolic tangent function.

$$S_p^i = \sigma(\sum \delta((x^i)^T w^j) v_p^i \tag{2}$$

4.3 Network Pruning

The goal of the pruning algorithm is to prune as many connections as possible. The penalty term identifies unnecessary connections by assigning smaller weights to them. Those connections can be removed without letting the accuracy drop. The penalty function is given by the sum of the following terms:

$$P(w,v) = \varepsilon_1 \Big(\sum_{j=1}^{h} \sum_{l=1}^{n} \frac{\beta (w_l^j)^2}{1+\beta (w_l^j)^2} + \sum_{j=1}^{h} \sum_{p=1}^{o} \frac{\beta (v_p^j)^2}{1+\beta (v_p^j)^2} \Big) \tag{3}$$
$$+ \varepsilon_2 \Big(\sum_{j=1}^{h} \sum_{l=1}^{n} (w_l^j)^2 + \sum_{j=1}^{h} \sum_{p=1}^{o} w_p^j)^2 \Big)$$

The first term indicates the accuracy and the second term indicates the complexity. $\varepsilon_1 > 0$ and $\varepsilon_2 > 0$ should be chosen to reflect the relative importance of the accuracy and complexity respectively. $\beta > 0$ is the slope of the error function near the origin.

The proof and derivation of the network pruning algorithm can be found in [15]. w_l^m denotes weights from input to hidden layer and v_p^m denotes weights from hidden to output layer. The criterion for removing the connections is the magnitude of their weights as can be seen from (5) and (6). The goal is to remove as many weights as possible as long as the accuracy stays within the acceptable limits. The algorithm forces the weights of the irrelevant links to converge to zero. The pruning algorithm is shown in Figure 2.

Figure 2. Pruning Algorithm

Inputs:

η_1: parameter which measures accuracy over the training examples

η_2: establishes criterion for weight removal

$(\eta_1 + \eta_2 < 0.5)$

Target classification rate

Set of weights from the trained network

Output:

Set of weights from the pruned network

1. Let η_1 and η_2 be positive scalars such that $\eta_1 + \eta_2 < 0.5$.
2. Pick a fully connected network. Train this network until a predetermined accuracy rate is achieved and for each correctly classified example the condition
   $$\max_p | e_p^i | = | S_p^i - t_p^i | \leq \eta_1 \quad (4)$$
   $\eta_1 \in [0, 0.5)$, is satisfied.
3. Let (w, v) be the weights of this network.
4. For each w_l^m if,
   $$\max_p | v_p^m w_l^m | \leq 4\eta_2 \quad (5)$$
   remove v_p^m from the network.
5. For each v^m if,
   $$\max_p | v_p^m | \leq 4\eta_2 \quad (6)$$
   remove v_p^m from the network.
6. If no weight satisfies (5) or (6), then remove w_l^m with the smallest product $| v_p^m w_l^m |$
7. Retrain the network. If classification rate of the network falls below an acceptable level, then stop. Otherwise go to Step 3

4.4 Feature ranking

The feature ranking method is based on the pruning algorithm explained above. The training phase employs a penalty function as a part of the objective function which assigns higher magnitude to important weights. After the completion of training phase the weights from hidden layer to output layer and the weights from input layer to the hidden layer indicate which links are going to be retained and which ones are going to get pruned. It is the magnitude of these weights that carries this information. Hence we propose to use the same information in a different way. From the above pruning algorithm it is safe to assume that the weights from hidden layer to input layer are going to be removed in the order of the sum of the product given by (5).

Sorting the inputs according to product of weights given by (5) and

Inputs:

Set of weights from the pruned network

Output:

Ranked list of features

1. Let w_l^m be the weights from input to hidden layer and v_p^m be the weights from hidden to output layer.
2. For each w_l^m , compute $|v_p^m w_l^m|$
3. Add these products up to get the sum sum_l^m
4. Sort the input nodes in the order of the product sum_l^m in the ascending order
5. Assign ranks to the inputs according to position of the input in the sorted list

Figure 3. Feature Ranking Algorithm Based on Sorting

ranking the attributes after training phase is the most intuitive method for ranking features. Although during the pruning phase the weights of the links change constantly due to the retraining. Hence the weights obtained

after the completion of training phase and the weights obtained after an iteration of pruning phase are different. We can observe the order in which the input nodes (features) get eliminated instead. We present the proposed feature ranking method in Figure 3 and Figure 4. Please note that we consider only the links from input layer to the hidden layer here as we want to rank the input attributes which correspond to the units in the input layer of the network. Given below are the two suggested methods for ranking the features. Method-1 is based on sorting the input nodes using condition (5) and method-2 is based on the order of removal of the links from the network.

In order to compare our results of feature extraction with some

Inputs:

Set of weights from the pruned network

Parameters needed for pruning

Output:

Ranked list of features

1. Apply pruning algorithm to the constructed network
2. Note the order in which the links in the network from input layer to hidden layer get pruned
3. When all the links from an input node to the hidden units in the hidden layer get removed, mark that node as removed
4. Assign a rank to this removed input node using the order of its removal from the set of input nodes

Figure 4. Feature Ranking Algorithm Based on Pruning

known measure for ranking attributes, we chose the information gain. The information gain measure is used in decision trees for selecting the attribute which would be the best classifier and hence would split a node of the decision tree. Information gain measures the expected reduction in entropy [21]. It is a statistical property that indicates how well a given attribute separates the training examples according to their target

classification. The entropy is commonly used in information theory for representing the impurity (uncertainty) of a random variable (see [22]). Given a collection S, containing positive and negative examples of some target concept, the entropy of S associated with this Boolean classification is,

$$Entropy(S) \equiv -p_+ \log_2 p_+ - p_- \log_2 p_- \qquad (7)$$

where p_+ is the proportion of positive examples and p_- is the proportion of negative examples in S. After defining entropy we can define information gain as the expected reduction in entropy caused by partitioning the examples according to the attribute A.

$$Gain(S, A) \equiv Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v) \qquad (8)$$

Where Values (A) is the set of all possible values for attribute A and Sv is the subset of S for which attribute A has a value v. More discussion of the information gain measure and its use in machine learning can be found in [21].

4.5 Rule Extraction

The pruning algorithm retains the links from the most important input attributes. It identifies the I-O relationships in the given piece of software, but it does not explain these relationships. The rule-extraction algorithm can explicitly state how an input affects the output. We use here the rule-extraction algorithm that is presented in [10], [16] and [20]. The pruning algorithm preserves salient connections of the NN. The goal of the rule-extraction is to express the I-O relationships retained by pruning in the form of if-then rules. One possible bridge between inputs and outputs of the pruned network are the hidden unit activation values. The input values and weights associated with them generate these hidden unit activation values. These hidden units activation values and the weights from hidden layer to output layer generate the final output. The

hidden units implement the hyperbolic tangent function and their output is always between -1 and 1. As hidden unit activation values are continuous, they need to be discretized before extracting the rules from the network.

The clustering algorithm given below in Figure 5, tries to cluster

Inputs:

Set of weights from the pruned network

ε : Parameter used to decided whether a value can be replaced while clustering the activation values

Output:

Discretized hidden unit activation values

For each hidden unit,

1) Let $\varepsilon \in (0,1)$,
2) Start with activation value α_0 of the first example in the training set
3) Cluster activation values (α_i) for the remaining examples if $|\alpha_i - \alpha_0| < \varepsilon$
4) Represent this cluster's activation value by the average of the activation values in the cluster
5) Select next α_0, repeat 3 and 4 for clustering until all activation values are clustered

Figure 5. Discretizing Hidden Unit Activation Values by clustering

hidden unit activation values into one interval as long as such replacement does not generate any conflicting results. The algorithm is applied to the activation values of all the hidden units for the different examples indicated by the index *i* below. We start with the activation value of the first hidden unit for the first example. We try to replace the

rest of the values with this value. We repeat this process for all the examples and for all the hidden units. When the clustering is over we get a few discretized values for every hidden unit instead of infinite continuous values. The choice of parameter ε depends on the accuracy after replacement. It can be increased as long as the accuracy of the network does not drop. We then proceed to rule-extraction.

The rule-extraction algorithm is divided into two parts as given in Figure 6. The first part focuses on the outputs generated with the possible combinations of discretized activation values for different hidden units. Once the outputs are known we find the inputs which can generate the discretized activation values, eliminate the activation values and thus link inputs to the outputs.

Inputs:

Discretized hidden unit activation values

Output:

Rules generated from the pruned network

1) Apply a clustering algorithm to find clusters of hidden node activation values.
2) Enumerate the discretized activation values and compute the network outputs. Generate rules that describe the network outputs in terms of discretized hidden unit activation values
3) For each hidden unit, enumerate the input values that lead to them and generate a set of rules to describe the hidden unit activation values in terms of inputs
4) Merge the two sets of rules obtained in the previous two steps to obtain rules that relate the inputs and outputs

Figure 6. Rule-Extraction Algorithm

4.6 Test Case Generation

The pruning phase retains the links of the most important attributes in the network. After the completion of pruning phase we know the most important input attributes but we do not know which values of every input attribute should be tested. If we generate test cases after the pruning phase we need to consider all possible attribute values. For example, we apply pruning to a piece of code with 10 input attributes and one output attribute. After pruning let A_1, A_2 and A_3 be the attributes identified as the most important attributes. Let 2 and 4 be the number of data values for the first two attributes. Let A_3 be a continuous attribute taking 100 different values. The number of test cases after pruning will be Cartesian product of these data values and they will be $2\times4\times100=800$.

The rule-extraction phase can help building equivalence classes for continuous attributes. For the above mentioned example if the extracted rules can partition the data values for A_3 into two intervals then we get two equivalence classes. Hence the number of test cases will be only $2\times4\times2=16$. When we explain implementation of our methodology for a 'Credit Card Approval System' we show exactly how the test cases are generated and listed.

5. A Case Study

The experiments conducted used a sample software application for a 'Credit Card Approval System' [12]. The specifications of this sample application were used to design the NN and the pseudo code from [12] was used to write the actual code which provided the training and testing data. The number of hidden layers, the number of hidden units in the hidden layer and the learning rate were determined by trial and error. We present the results of the following phases for the credit card approval program:

- Pruning
- Feature Ranking by method 1
- Feature ranking by method 2
- Comparison of the results with information gain measure
- Rule-extraction

- Test case generation after pruning
- Test case generation after rule-extraction

Table 1. I-O attributes for credit-card approval system

Attribute	**Meaning**		**Value**	**Type**
Unique-Descriptor	Identifies every record		Non-Continuous	Apha-Numeric
Citizenship	0: American 1:Other	Input	Non-Continuous	Numeric
State	0:Florida 1:Other	Input	Non-Continuous	Numeric
Age	1-100	Input	Continuous	Numeric
Income	0:Not Steady 1:Steady	Input	Non-Continuous	Numeric
Region	0-6 (Different regions in US)	Input	Non-Continuous	Numeric
Income Class	0-3 (Divided into ranges) 0 if income p.a. <$10k 1:if income p.a. ≥$10k 2 if income p.a. <$25k 3:if income p.a. ≥$50k	Input	Non-Continuous	Numeric
Marital Status	0:Unmarried 1:Married	Input	Non-Continuous	Numeric
Number of dependents	0-4	Input	Non-Continuous	Numeric
Credit Approved	0:No 1: Yes	Output	Non-Continuous	Numeric

5.1. The sample application

The specifications of a sample software application were used for setting the different parameters of the neural network and to generate training and test data. The 'Credit Card Approval System' recommends whether or not a credit card should be granted to a customer depending on his/her age, income, citizenship etc. which are the inputs to this system. As indicated in [12], this program can be considered representative of a wide range of business applications. *Table 1* gives the detailed description of the software attributes, their types and values.

5.2. Constructing Neural Network

The first input attribute in this application is the unique identifier, which does not affect the output and hence is not considered while building the NN. So we consider only the next eight relevant inputs for building the NN. The inputs of the sample applications do not have a uniform representation in terms of type (continuous vs. discrete) and values (numeric vs. nominal). NNs can work only with numeric data but typical business applications include nominal attributes as well (e.g., *region*). The raw data obtained from the application hence required some preprocessing.

We mapped the values of multi-valued input attributes (such as *age*) to the range between 0 and 1 so that the NN can process the values uniformly. We normalized the data according to the maximum and minimum possible values for each attribute. For attributes like *region* which were converted into enumerated values we took the minimum and maximum from the enumerated values for normalization. The output is binary indicating whether the credit has been approved or not. No pre-processing was required for the output. We used eight non-computational input units for the eight relevant input attributes and one output computational unit. The number of hidden units and the hidden layer was determined by trial and error.

5.3 Training, pruning and rule-extraction

We used a NN with eight input nodes corresponding to eight input attributes one hidden layer having four units and one output layer with just one output node corresponding to one binary output (credit approval). The size of the training data and test data set was 1000 examples each. The input data was generated randomly for each of the inputs. The input was then fed to the code which then generated the outputs. The inputs were preprocessed and normalized as explained above before feeding them to the NN.

The weights were generated randomly within the range (-0.5, 0.5). The learning rate was set to 0.5. The values chosen for $\varepsilon_1, \varepsilon_2, \beta$ were 0.1, 0.0001 and 10 respectively. Equation (1) and (3) together give the objective function used in the back-propagation. As the back-propagation advanced through the epochs the difference between the values of the objective function for the previous iteration and the current iteration over the entire set of training examples was calculated. This difference was used as the stopping criterion. If the difference dropped below the acceptable value (set to -0.001) the training process was terminated.

The pruning phase had three parameters to be tuned. Parameter η_1 checked, whether the difference between the actual output and the expected output in every classified example is within the acceptable limits (set to 0.25). The second relevant parameter to pruning η_2 indicating the threshold for the weight removal was set to 0.2. Before starting the next iteration the network from the previous iteration was saved. If the next iteration dropped the accuracy of the network below an acceptable limit the network from the last iteration could be retrieved. We set the acceptable level of accuracy to 95% that is if after removing a link the accuracy was less than 95% we decided not to remove the link, retrieve the network from the previous iteration and terminate the pruning. The parameter ε, which is the acceptable difference between the hidden unit activation values before replacement used in discretization was determined iteratively. Starting from 0.2 if the replacement did not affect the accuracy of the rule generated a higher value was tried the next time.

5.4. Test Case Generation

The 'Credit Card Approval System' has 8 inputs. *Table 2* lists the number of data values / intervals taken by each of these attributes. Except for the attribute *age*, all the attributes are discrete and hence the number of data values can easily be enumerated. *Age* being a continuous attribute varying from 1 to 100, we discretize it to 10 intervals. We can calculate the total number of test cases by taking a Cartesian product of the corresponding data values. The number of combinatorial test cases is therefore 22400. We list below the test cases obtained after pruning and after rule-extraction.

The weight decay back-propagation algorithm needed 1500 epochs to provide an accuracy of 99-100% on training data and 98-100% accuracy on test data.

We ran the pruning iteratively on the data and noted the results of the pruning and the feature ordering performed on the 'Credit Card Approval System'. The order in which the weights were removed, the ordering of features using the order of weight removal and the ordering of features by sorting the weight from input layer according to condition (5) was observed and noted. Consistent ranking of attributes 2 (*age*) and 4 (*region*) for all the training sets with both methods was obtained. Age was always ranked as the best attribute while *region* was ranked as the second best. It was observed that the order of removal of the rest of the attributes varied between the runs.

The last link which is chosen for removal is in fact retained in the network. The reason for this is, this link is chosen for weight removal but as it drops the accuracy of the network below the acceptable limit it is retained. Here the weights from the last saved iteration are stored and retrieved. The weights retained in the network appear in the last column of the table and they appear in bold.

We compare the ordering obtained by our methods with the ordering obtained by the information gain measure in *Table 3* where the attributes are sorted in the ascending order of the information gain. This means the attributes appearing up in the list have lower information gain than attributes which appear at the bottom of the list. The information gain measure assigns the highest importance to *region* and *age* respectively.

Table 2. Possible data values and numbers given to the attributes of 'Credit Card Approval System'

	Possible Values/ Range	Number of values/ Intervals	Numbers Given
Citizenship	0/1	2	0
State	0/1	2	1
Age	1-100	10	2
Steady Income	0/1	2	3
Region	0/1/2/3/4/5/6	7	4
Income Class	0/1/2/3	4	5
Number of dependents	0/1/2/3/4	5	6
Married/ Unmarried	0/1	2	7

These are the two attributes receiving the highest ranking by both feature extraction methods. This does not match the results of feature ranking exactly. The information gain assigns a rank to every attribute in absence of the other attributes as against our feature ranking methods which rank an attribute in presence of the other attributes. Consequently, the information gain fails to take into consideration the interaction between various attributes as opposed to our feature ranking methods.

In *Table 3* we list the results of conditional mutual information calculated by IFN [5 & 6]. Conditional mutual information [22] is an information-theoretic measure, which does take into account the interaction between input attributes. At each step, the IFN algorithm incrementally chooses the next input attribute, which has the highest conditional mutual information with respect to other attributes, which

have not been included in the network yet. The conditional mutual information for *region* which is the first attribute selected by IFN (rank = 1) is very close to information gain for the same attribute in *Table 4*. Though *age* is only the second attribute selected by the algorithm (rank = 2), its conditional mutual information in the second iteration is slightly higher than the conditional mutual information of *region* in the first iteration, which explains, why *age* was preferred over *region* by our pruning (backward elimination) methods. IFN discards the rest of the attributes as their conditional mutual information is statistically insignificant.

Table 5 shows the testing accuracy before and after pruning for each test dataset. After training phase the accuracy on test data is 98-100%. The potential increase in the accuracy which is already so high cannot be more than 1%-2%. Pruning removes the unnecessary connections; once these connections are removed the network can generalize better on the test data and hence the testing accuracy was improved or remained the same in almost all the cases.

Table 3. Ranking by IFN conditional mutual information gain of 'Credit Card Approval System'

Number	Attribute	Conditional Mutual Information	Rank in the Network
1	State	0	-
0	Citizenship	0	-
5	Income Class	0	-
7	Marital Status	0	-
3	Steady Income	0	-
6	Number of dependents	0	-
2	Age	0.501	2
4	Region	0.484	1

Table 4. Possible data values and numbers given to the attributes of 'Credit Card Approval System'

Serial Number	Attribute	Information Gain	Rank
1	State	0	8
0	Citizenship	0.0001	7
5	Income Class	0.003425	3
7	Marital Status	0.0003466	5
3	Steady Income	0.000845	6
6	Number of dependents	0.194798	4
2	Age	0.276645	2
4	Region	0.486261	1

Table 5. Comparison of test accuracy before and after pruning

	Accuracy on Test Data before Pruning	Accuracy on Test Data after Pruning
Set 1	99	100
Set 2	99	99
Set 3	98	99
Set 4	99	99
Set 5	99	100
Set 6	99	100
Set 7	99	100
Set 8	98	100
Set 9	99	99
Set 10	99	99

Figure 7 shows the NN before pruning. It had 32 links from input layer to the hidden layer and another 4 links from hidden layer to the output layer. The entire network had 36 links. In Figure 8 the network after pruning has been shown. As can be seen from Figure 8 the size of the network reduces remarkably after pruning. There are just two links from the input layer to the hidden layer and just two more from the hidden layer to the output layer. Two hidden units have been removed completely.

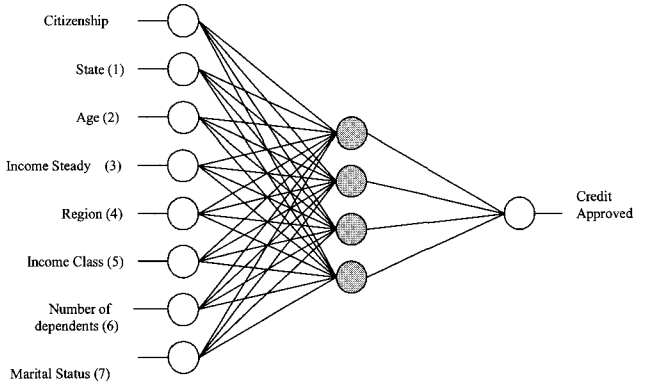


Figure 7. Neural network used for 'Credit Card Approval System'

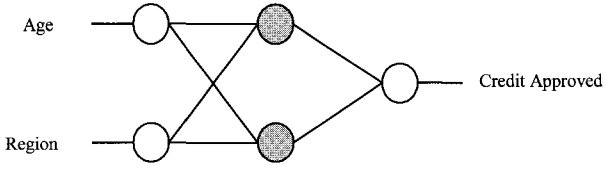


Figure 8. Neural Network after pruning

Since *age* and *region* have been identified as the most important attributes we should build the test cases with just these two attributes. As given in *Table 2*, number of equivalence classes on these two attributes is 10 and 7 respectively. Now we need only 70 test cases to test this program. This is a huge reduction in the number of test cases as they

have been reduced from 22400 to 70. Since attribute *age* has 10 intervals as its equivalence classes we choose a representative value for each of these intervals and use that for listing the test cases. For example, 25 is the representative value for interval 21-30. Any value from the interval can be chosen for generating test cases. Now we present the results of the rule-extraction phase with the list of reduced test cases.

Figure 9 shows one of the pruned networks. This network achieved 100% accuracy on test data after pruning and is the representative network used for rule-extraction. The weights from the two inputs to the two hidden units, the values of the bias shown and the weights associated with them are given below in Figure 10.

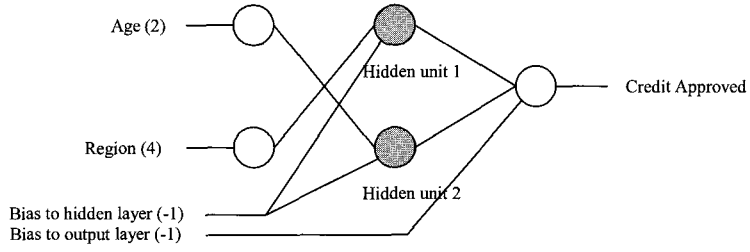


Figure 9. Neural network used for rule-extraction

Weight from 'age' to hidden unit 2	→	61.833301
Weight from 'region' to hidden unit 1	→	19.496375
Weight from hidden unit 1 to output	→	-7.69731
Weight from hidden unit 2 to output	→	8.392605
Weight of bias to hidden unit 1	→	15.221573
Weight of bias to hidden unit 2	→	10.445275
Weight of the bias to the output	→	6.337605

Figure 10. Rule-Extraction Algorithm

The activation values of the hidden units were first discretized. There were two values for hidden unit corresponding to *region*: -1 and 1. The two values obtained for activation unit corresponding to *age* were -1 and 1 as well. Once the values were discretized, output was calculated for various combinations of these activation values. The next step was to find the input values for which these particular activation values could be generated. *Region* being a discrete attribute, it was easy to find the input values corresponding to the activation values. After computing the different possible values of attribute *region* and the activation values for those inputs, we observed that the activation values for *region*= 0, 1, 2, 3, 4 were clustered as -1 and those for *region*= 5, 6 were clustered as 1. For the attribute *age*, a different procedure was followed. We sorted the different values for attribute *age* and observed the corresponding activation values. We observed that activation values in interval (-1, -0.33) corresponded to inputs 1-17 and the rest of them that is starting from 0.27,1 corresponded to *age* 18-100. So *age* 18 was the threshold used for building two equivalence classes on attribute *age*. We combined these two results to generate the following rule,

If (age $\geq$ 18) and (region = 0 or 1 or 2 or 3 or 4)
    Credit Approved= 'yes'
Else
    Credit Approved= 'no'

The accuracy of this rule on the test data was 100%. The activation values of the hidden unit corresponding to attribute *age* clearly partitioned the range of this attribute into two intervals. Hence we get two equivalence classes on *age* and they would be,

Age 1: [1-17]
Age 2: [18-100]

Similarly for attribute *region* we clearly get two equivalence classes with two activation values viz. -1 and 1 and they are,

Region 1: [0, 1, 2, 3, 4]
Region 2: [5, 6]

We can use this information to further reduce the test cases. For attribute *age*, let 15 be the representative value of class 1 and 50 be of class 2. For attribute *region*, let 2 represent class1 and 5 represent class 3. Given below in *Table 6* is the reduced set of test cases. This is a reduction of 66% (from 12 to 4) vs. the results produced by the info-fuzzy test reduction method in [5] for the same case study. The reason is that the info-fuzzy method does not provide the capability to automatically identify equivalence classes in discrete attributes such as *region*.

Table 6. List of Test Cases

Test ID	Region	Age	Output
1	2	15	0
2	2	50	1
3	5	15	0
4	5	50	0

6. Conclusions

In this paper we have presented an automated approach to input-output analysis of the tested software. The approach is based on construction, training, and pruning of an artificial neural network (ANN). We have been able to identify the most significant attributes which influence an output of a sample business application. Consequently, we have shown a significant decrease in the number of test cases as compared to an alternative method of automated test case reduction (info-fuzzy network). The proposed method is a black-box approach. No specification details or source code are needed for using it. The NN is nothing but a mathematical model of the software and hence can be used as an automated oracle to predict the outputs of the test cases generated.

The rule-extraction not only reduces the number of test cases but also helps identifying equivalence classes for nominal attributes.

When the ranking of features is available it is possible to stop the testing effort at any time by focusing on test cases involving the attributes with higher scoring. This gives valuable guidelines to prioritizing test cases when the available testing resources are limited.

We plan to extend the methodology to testing software with multiple outputs. It would be interesting to see if we obtain some useful information after the completion of the training phase itself which could help us to predict a better and more accurate ordering of features.

Acknowledgements

This work was partially supported by the National Institute for Systems Test and Productivity at University of South Florida under the USA Space and Naval Warfare Systems Command Grant No. N00039-01-1-2248.

References

[1] C. Jorgensen, *Software Testing: A Craftsman's Approach: Paul* CRC Press, 2002.

[2] C. Kaner, J. Falk, and H.Q. Nguyen, *Testing Computer Software*, Wiley Computer Publishing, 1999.

[3] C. Yang, and L.L. Pollock, *Identifying Redundant Test Cases For Testing Parallel Language Constructs*, Advanced Telecommunication/ Information Distribution Annual Conference, College Park, MD , January 1997.

[4] P.J. Schroeder and B. Korel, "Black-Box Test Reduction Using Input-Output Analysis", *Proc. of ISSTA* '00, 2000: p. 173-177.

[5] M. Last and A. Kandel, "Automated Test Reduction Using an Info-Fuzzy Network", to appear in Annals of Software Engineering, Special Volume on *Computational Intelligence in Software Engineering*, Kluwer, 2003.

[6] O. Maimon and M. Last, *Knowledge Discovery and Data Mining-The Info-Fuzzy Network* (IFN) Methodology, Kluwer Academic Publishers, Massive Computing, Boston, December, 2000.

[7] T. Kohonen, *An introduction to Neural Computing*, Neural Networks, 1988. 1: p. 3-16.

[8] R. Andrews, J. Dierich, and A. Tickle, *Survey and critique of techniques for extracting rules from trained Artificial Neural Network*, Knowledge-Based Systems, December 1996, 8(6).

[9] D. Bahler and B. Stone, *Neural Models and Extracted Rules for Knowledge Discovery in Predictive Toxicology,* available at: [http://citeseer.nj.nec.com/469862.html]

[10] R. Setiono, *Extracting rules from pruned neural network for breast cancer diagnosis*, Artificial Intelligence in Medicine, 1996(8): p. 37-51.
[11] C. Anderson, A. von Mayrhauser, R. Mraz, *On the use of NN to guide software testing activities,* Proceedings of ITC '95, the International Conference, October, 1995, 21-26.
[12] M. Vanmali, M. Last and A. Kandel, "Using a Neural Network in Software Testing Process", *International Journal of Intelligent Systems*, 2002, 17: p. 45-62.
[13] M. Vanmali, *Using a neural network in the software testing Process*, 2002, Master Thesis. University of South Florida.
[14] R. Reed, "Pruning Algorithms - A Survey", *IEEE Transactions on Neural Networks*, September 1993, 4(5).
[15] R. Setiono, *A penalty-function approach for pruning feed-forward neural network*, Neural Computation. 9(1): p. 185-204.
[16] R. Setiono and H. Liu., "Effective Data Mining Using Neural Networks", *IEEE transactions of Knowledge and Data Engineering*, December 1996, 8(6).
[17] E.D. Karnin, "A Simple Procedure for Pruning Back-Propagation Trained Neural Network", *IEEE Transactions of Neural Networks*, June, 1990, 1(2).
[18] G. Towell and J. Shavlik, "The extraction of refined rules from knowledge based neural networks", *Machine Learning*, 1993. 3: p. 71-101.
[19] M.W. Craven and J.W. Shavlik, "Using sampling and queries to extract rules from trained neural networks", Machine Learning: *Proceedings of the Eleventh International Conference*, San Francisco, CA, USA, 1994.
[20] R. Setiono and H. Liu, "Understanding Neural Networks via Rule-extraction", Proc. Of *IJCAI'95*, 1995.
[21] T.M. Mitchell, *Machine Learning*, McGraw-Hill Press, 1997.
[22] T.M. Cover, *Elements of Information Theory*, New York: Wiley, 1991.
[23] M. Last, H. Bunke, and A. Kandel, "A Feature-Based Serial Approach to Classifier Combination", *Pattern Analysis and Applications* (PAA), No. 5, pp. 385-398, 2002.

CHAPTER 5

THREE-GROUP SOFTWARE QUALITY CLASSIFICATION MODELING USING AN AUTOMATED REASONING APPROACH

Taghi M. Khoshgoftaar and Naeem Seliya

Empirical Software Engineering Laboratory
Dept. of Computer Science and Engineering
Florida Atlantic University, Boca Raton, FL 33431, USA
E-mail: {taghi, nseliya}@cse.fau.edu

Effectiveness of software testing and inspection endeavors can be increased dramatically if software quality estimation was available prior to system tests and operations. Such a prediction can allow focused utilization of the testing and quality improvement resources. Software quality classification models are used to classify system modules into risk-based classes or groups. In cases of limited software testing and quality improvement resources or when the risk factor (faults) varies considerably, three group models are preferred over two group classification models.

An innovative algorithm that circumvents the complexities involved in a pure three-group software quality classification technique is presented. This technique utilizes any two-group classification technique three times in order to yield three risk groups, i.e., high-risk, medium-risk, and low-risk. We evaluate the algorithm using case-based reasoning (CBR), a computational intelligence system that uses an automated reasoning approach. Our first attempt at the application and validation of the algorithm is presented with a case study. Software metrics and fault data are collected from a real time data communications system. The calibrated three group model is evaluated against the one obtained by discriminant analysis. Promising results in terms of classification accuracy and model

stability were observed. Significant research is needed for further validating the usefulness of the proposed three group classification algorithm as an aid for effective software testing and reliability improvement.

1. Introduction

An important goal of software testing and quality improvement endeavors for a software system is achieving high reliability with minimal maintenance [1]. This is especially important in high-dependence software systems, such as mission-critical and telecommunications systems. One may argue that the use of exhaustive software testing with all possible use-case scenarios may be the best option. However, practically speaking almost all software projects have pre-assigned monetary and product development resources, which are finite and limited. Prior to system operations, project managers are more than often required to decide which system modules should be targeted for expending software testing and reliability improvement activities.

The effectiveness of analysis and testing of software systems can be improved significantly, if the team were to benefit from an early (during system tests or prior to deployment) identification of modules according to a quality (or risk) factor. Software metrics-based quality classification models can be designed to provide such an early prediction [19,31]. Researchers have developed and applied effective algorithms and techniques to classify modules according to their associated risk, such as number of faults. A few of such classification techniques include, case-based reasoning [14,26], decision trees [10,12], neural networks [15,24], logistic regression [8,30], optimized set reduction [4], fuzzy logic [6,34], discriminant analysis [19], and genetic programming [18].

Software quality classification (to be referred to as SQC) models, usually classify (using independent variables) modules or observations into two groups, such as high-risk or fault-prone and low-risk or not fault-prone [5,20,21,23,27]. Such two-group models assume that enough software testing and quality enhancement resources are available for all the modules identified as high-risk. However, the fact remains that such project resources are almost always limited and may not be applied to all

the modules predicted as high-risk. Moreover, among the modules identified as high-risk, the associated risk factor may vary considerably, limiting the usefulness of a two-group classification model.

In cases where a more effective resource utilization and improved direction for targeting high-risk modules is needed, a three-group SQC model should be preferred. For example, modules can be classified as Red (high-risk), Yellow (medium-risk), and Green (low-risk) [35]. Based on such a risk-based prediction for modules currently under development, the software quality assurance team can then deploy effective reliability enhancement efforts. The main aim of a three-group model is to predict all the high-risk modules correctly. Therefore, all modules predicted as Red, should be subjected to extensive quality improvement efforts. In contrast, those predicted as low-risk (Green) should not be reviewed at all. Furthermore, the modules predicted as Yellow might, depending on resource availability, be subjected to quality improvements using only a fraction of the effort and resources used for the Red modules.

Moving from a simple two-group classification to a three-group classification problem, involves issues that are often difficult to address. A three-group classification involves two threshold values (as compared to one for a two-group), which inherently are difficult to estimate. An early investigation at performing a three-group classification by Basili et al. [3], describes a procedure for modeling the difficulty of making a change to software, i.e., easy, undecidable, or difficult. Their approach used a subjective variable called *difficulty index*, which depended on the person assessing the difficulty. Furthermore, the decision boundaries between the three groups were intentionally optimized to yield better results, contradicting the simplicity of the approach.

Another problematic issue is the complexity of classification problems, which increases dramatically as the number of groups increases, i.e., three-group classification problems. For example, Boolean discriminant functions have been used to classify modules into two groups [28,29], depending on whether the associated function yields a true or a false. If a similar approach was to be followed for a three-group classification a *trinary* discriminant function would be needed, which by

nature involves more complex computations. Consequently, the simplicity of a two-group model is often preferred over a three-group model, even if the latter yields better solutions and is more appropriate.

This study presents and evaluates an innovative three-group classification algorithm that circumvents the need for a pure three-group classification technique. The foundation of our approach involves applying a two-group classification technique three times on a given data set. The proposed algorithm can therefore utilize any existing two-group technique, such as neural networks [15], case-based reasoning [14], logistic regression [8] and genetic programming [18], for the given three-group classification problem. We demonstrate the algorithm by calibrating three-group SQC models using case-based reasoning (CBR) [14]: a computational intelligence technique that employs an automated reasoning process in order to resolve new problems [17].

CBR has been proven to be very useful in numerous software engineering domains, including software cost estimation [33,36], software quality estimation [14], and software design and reuse [25]. It is especially useful in an environment where there is limited knowledge and when an optimal solution in not known. Khoshgoftaar et al. have proposed and investigated two-group classification techniques using CBR [14,26]. This study extends our on-going research with CBR-based SQC by applying it to the proposed algorithm. Furthermore, both the two-group and three-group SQC techniques using CBR have been implemented in a user-friendly empirical research tool, SMART: the Software Measurement and Analysis Reliability Toolkit [11].

The case study used to illustrate the application of our three-group SQC technique consists of software measurement and fault data collected from a commercial real-time data communications system, written in C++. The data consisted of both, procedural and object-oriented software measures. The dependent variable, number of faults, was reported by the source code control system, which tracked the software system in our study.

A SQC model is usually associated with its misclassification error rates. In a three-group model, such as *Red*, *Yellow*, and *Green*, any one of the three classes could be misclassified as either of the two remaining

classes, thus giving a total of six possible misclassifications. Evaluating such a model based solely on its misclassification rates is not practical. This is because, the costs of rectifying each of the six errors are not similar. For example, a low-risk (Green) module if misclassified as a high-risk (Red) module would entail ineffective code reviews and testing. On the other hand, if a Red module is misclassified as Green implies missed opportunities of correcting high-risk modules prior to deployment: leading to expensive on-site repairs and downtime costs.

Comparing three-group SQC models with each other may be a daunting task, because each of their six error rates could vary (low or high) as compared to their respective counterparts. A commonly used solution for the disparate cost issue is to use the overall error rate as the model-performance measure. However, it usually implies the use of equal costs for all the misclassifications, leaving the issue unresolved. A singular performance measure that considers the different costs of misclassifications is thus warranted. This study utilizes such a measure, "expected cost of misclassification" (ECM) that can be used to evaluate the performance of three-group classification models. Khoshgoftaar et al. first introduced the use of ECM in the context of controlling overfitting in two-group SQC trees [12].

Szabo and Khoshgoftaar present a pure three-group SQC modeling technique using discriminant analysis in their recent work [35]. The model calibrated using the proposed technique is evaluated against the three-group discriminant analysis model. The classification accuracy of the models is evaluated using ECM. Furthermore, since ECM is dependent on the costs of misclassifications, we performed a sensitivity analysis on the two modeling techniques in order to determine their model-stability. The actual misclassification costs are not known until very late in the system life cycle. Therefore, different ECM values are computed by varying the misclassification costs to encompass a broad spectrum of values.

Promising results in terms of good classification accuracy and model-stability were observed for the three-group model calibrated using the proposed algorithm. This study is our first attempt at illustrating the proposed algorithm. However, significant research in the near future will

further validate the usefulness of the proposed three-group classification algorithm as an aid for effective software testing.

The layout of the remainder of this study is as follows. Section 3 presents the proposed algorithm. In Sections 2 and 4 we present a brief overview of the basic concepts of SQC modeling with CBR and discriminant analysis, respectively. Section 5 discusses our empirical investigations including modeling methodology and results. Lastly, we conclude with Section 6, by summarizing our findings and discussing our future research directions.

2. Case-Based Reasoning

A CBR system is an expert system that aims at finding solutions to a new problem based on an automated reasoning approach [17]. Such a solution is evaluated based on past experience, represented by cases (with similar attributes) in a case library. Generally speaking, a CBR system consists of a case library, a solution process algorithm, a similarity function, and the associated retrieval and decision rules.

Useful and effective SQC models can be built with CBR using few primitive, as well as a large number of complex metrics [14]. Consequently, data collection efforts can be minimized, leading to the practical usefulness of CBR-based models. As compared to other software quality estimation modeling techniques CBR has several advantages, such as *case alertness*, *user acceptance*, *model adaptivity*, *scalability*, and *interpretation*.

When a new case is outside the bounds of current experience, CBR systems can *alert* users. This is advantageous because, instead of guessing a solution an answer of "I don't know" is usually better. Many other techniques require frequent re-calibration to adjust model parameters in lieu of new or updated information. However, cases can be added or updated from the case library of CBR systems, without the need for extensive model re-calibration. Furthermore, fast retrieval of cases continues to be of practical advantage for CBR systems, since they are *scalable* to very large case libraries. Unlike other modeling techniques such as neural networks, CBR-based models are not "black boxes".

Users can easily be convinced that a given solution was derived in a reasonable way, and hence, the CBR system lends itself to *user acceptance.*

2.1 Two-Group Classification with CBR

The working hypothesis of our previously developed two-group classification approach using CBR [14,26] is that a module (new case) currently under development is likely to have the same risk factor (high or low) as previously developed modules (case library) with similar attributes (predictors). The two-groups consisted of the fault-prone (*fp*), i.e., high-risk, and not fault-prone (*nfp*), i.e., low-risk, modules. A project-specific threshold value, such as number of faults (risk factor), separates the two groups (or classes).

A CBR system can function as a SQC model, prior to unit testing and system operations. Therefore, a good solution algorithm is one that provides a module class assignment that turns out to be correct after all the fault data is known. If each case in the case library has known attributes and class membership, then given a case with unknown class, we estimate its class to be the same as that of the most similar case(s), where similarity is determined in terms of the module attributes.

In a two-group classification model a Type I error occurs when a *nfp* (or low-risk) module is misclassified as *fp* (or high-risk), whereas a Type II error occurs when a *fp* module is misclassified as *nfp*. Practically speaking the costs of the two misclassifications, Type I and Type II, are not equal. Type II errors are more expensive and involve severe consequences, such as repairs at remote sites and damaging an organization's reputation. On the other hand, Type I errors are relatively less expensive, and may involve in-effective reviews and software testing.

A classification model may be sensitive to the ratio of the costs of the Type I and Type II misclassifications, i.e., C_I and C_{II}, respectively. This is because in practice actual values for C_I and C_{II} are unknown during modeling. And since $C_I \neq C_{II}$, the use of equal costs of misclassifications during model calibration is not realistic. Moreover, other factors besides

cost ratio may determine the best balance between the Type I and Type II error rates. For example, when the proportion of *fp* modules (as compared to *nfp*) is very small, and Type II misclassifications have relatively severe consequences, one may prefer *equal* misclassification rates [16].

The cost ratio, C_I / C_{II}, can be varied during modeling to obtain the preferred balance between the two error rates. In many software projects, costs of misclassifications, and similarly, prior probabilities of the *fp* and *nfp* classes, might be unknown or difficult to estimate. Thus, a generalized rule that does not require the knowledge of the misclassification costs and prior probabilities is needed. Khoshgoftaar et al. introduced such a generalized classification rule that minimizes the expected cost of misclassification [9]. More recently, Khoshgoftaar et al. proposed two similar generalized classification rules in the context of SQC modeling with CBR [14,26].

The case library consists of well-known project data from previously developed (similar) systems, and contains all relevant information pertaining to each case. In the context of a two-group SQC problem, such cases are composed of a set of *independent* variables ($\mathbf{x}_i$) and a response or *dependent* variable (y_i), i.e., risk class. A solution process algorithm uses a similarity function to measure the relationship between the new case and those in the case library. Consequently, relevant cases retrieved from the case library are then used to estimate a solution for the new case.

A *similarity function* determines, for a given case, the most similar cases from the case library (*fit* or *training* data). The function computes the distance d_{ij}, between the current case $\mathbf{x}_i$, and every other case $\mathbf{c}_j$ in the case library. Three types of similarity functions, i.e., the *City Block*, *Euclidean*, and *Mahalanobis* distances, were investigated in our research with CBR-based SQC modeling [14,26]. Software metrics collected during the pre-testing phases are often measured in varying ways and could contain a variety of ranges and scales. Therefore, before using the data for modeling purposes, it should be *standardized* (or *normalized*) [2].

The cases with the smallest possible distances are of primary interest, and the set of similar cases forms the set of *nearest neighbors*, N. Model parameter n_N, represents the number of the best (most similar to current case) cases selected from N for case analysis and class estimation. n_N can be varied during model calibration to obtain different models. Once n_N is selected, a classification technique, i.e., either *majority voting* or *data clustering* is used to estimate the risk class of the current case.

2.1.1 Mahalanobis Distance Similarity Function

An alternative to the Euclidean distance, this distance function is used when the independent variables are *highly correlated.* The Mahalanobis distance is a very attractive similarity function to implement because it can explicitly account for the correlation among the attributes [13], and the independent variables do not need to be standardized or normalized. It is given by:

$$d_{ij} = (\mathbf{c}_j - \mathbf{x}_i)' S^{-1} (\mathbf{c}_j - \mathbf{x}_i) \tag{1}$$

here, (') implies transpose, S is the variance-covariance matrix of the independent variables over the entire case library, while S^{-1} is its inverse. We use this similarity function to illustrate the proposed three-group classification algorithm with CBR.

2.1.2 Data Clustering Classification Rule

In this classification method, the case library is partitioned into two clusters, *fp* and *nfp*, according to the class of each case in the fit data set. For an unclassified case $\mathbf{x}_i$, $d_{nfp}(\mathbf{x}_i)$ is the average distance to the *nfp* nearest neighbor cases, and $d_{fp}(\mathbf{x}_i)$ is the average distance to the *fp* nearest neighbor cases. The number of nearest neighbor cases, i.e., n_N, to be used for analysis, can be varied as a model calibration parameter.

The generalized *data clustering* classification rule used to estimate the class of the unclassified case ($\mathbf{x}_i$) is given by:

$$Class(\mathbf{x}_i) = \begin{cases} fp, & \text{if } \dfrac{d_{nfp}(\mathbf{x}_i)}{d_{fp}(\mathbf{x}_i)} \geq c \\ nfp, & \text{otherwise} \end{cases} \tag{2}$$

The right hand side of the inequality, c, is the cost ratio of the two misclassification error rates, i.e., C_I / C_{II}. Since, the actual costs of misclassification are unknown until very late in the life cycle, c represents the *modeling cost ratio* which can be empirically varied as per the needs of a given software system. The classification of the current case would then depend on whether or not the ratio, $d_{fp}(\mathbf{x}_i) / d_{nfp}(\mathbf{x}_i)$, exceeds the chosen value of c. We used this classification rule to illustrate the proposed three-group classification algorithm with CBR.

2.2 Three-Group Classification Algorithm

The algorithm presented in this section, avoids the hassle involved in pure three-group classification methods, such as discriminant analysis. To reiterate, a three-group SQC model requires the estimation of two threshold (or critical) values in order delineate the three classes. These threshold values are usually dependent on past experience with similar projects in similar environments. Furthermore, the computational complexity increases dramatically as the number of groups in a classification problem increases.

The proposed algorithm circumvents the above-mentioned issues by utilizing the more commonly used two-group classification technique three times. Therefore, a need to develop a direct three-group classification model is avoided. To demonstrate the approach we use our previously described two-group technique using CBR. However, the technique is not tied into CBR-based modeling. It can be customized to suit any existing two-group SQC technique, such as neural network, logistic regression, etc.

In our case study presented in this research, the number of faults detected in a software module determines its risk class or risk factor. Consequently, the case library (fit or training data) is partitioned into three groups, i.e., Green (low-risk), Yellow (medium-risk), and Red

(high-risk). Let us denote the two threshold cut-off values for number of faults (denoted as *Faults*) as *c_low* and *c_high*. A module with *Faults* greater than *c_high* will belong to the Red group, whereas a module with *Faults* less than or equal to *c_low* will belong to the Green group. All other modules will belong to the Yellow group.

Table 1. Misclassification for a Three-Group Model

Error	Cost	Number	Description
Type_GR	C_{GR}	n_{GR}	Green classified as Red
Type_GY	C_{GY}	n_{GY}	Green classified as Yellow
Type_YR	C_{YR}	n_{YR}	Yellow classified as Red
Type_YG	C_{YG}	n_{YG}	Yellow classified as Green
Type_RY	C_{RY}	n_{RY}	Red classified as Yellow
Type_RG	C_{RG}	n_{RG}	Red classified as Green

The following steps illustrate the implementation of our proposed three-group classification algorithm using CBR. The entire process is automated by our modeling tool, SMART [11], which facilitates the empirical variation of the modeling parameters, n_N and c.

1. Using a similarity function, compute the distance of each unclassified case in the test data set with respect to all the Green and Red cases in the fit data set. Given the computed distances, parse the test data set into two groups, and denote them as *Test-I* and *Test-II*. The generalized data clustering classification rule [16] used for the parsing is given by:

$$Class(\mathbf{x}_i) = \begin{cases} Green & \text{if } \dfrac{d_{Red}}{d_{Green}} > \dfrac{C_{GR}}{C_{RG}} \\ Red & \text{otherwise} \end{cases} \quad (3)$$

where, d_{Green} and d_{Red} represent the average distances to the Green and Red groups (fit data), respectively. C_{RG} and C_{GR} are the misclassification costs of the Type_RG and Type_GR errors, respectively. Table 1 summarizes the notations used for misclassification error types, their costs and number of misclassifications. The two-group model built in this step is denoted as Model_GR. It should be noted that the two groups parsed in Equation (3), are not purely Green or Red. We shall see shortly that both Test-I and Test-II may contain modules that are Yellow.

2. For each of the modules parsed into the Test-I group, compute (same similarity function as in step 1) the distances with respect to all the Green and Yellow cases in the fit data. Subsequently, parse the Test-I group into two sub-groups, and denote them as *Test-Green* and *Test-Yellow_I*. The classification rule to achieve the same is given by:

$$Class(\mathbf{x}_i) = \begin{cases} Green & \text{if } \dfrac{d_{Yellow}}{d_{Green}} > \dfrac{C_{GY}}{C_{YG}} \\ Yellow & \text{otherwise} \end{cases} \quad (4)$$

where, d_{Green} and d_{Yellow} represent the average distances to the Green and Yellow groups (fit data), respectively. The two-group model built in this step is denoted as Model_GY, as it classifies modules as either Green or Yellow. At this point all the Green modules of the test data set have been either classified correctly or misclassified.

3. For each of the modules parsed into the Test-II group, compute (same similarity function as in steps 1 and 2) the distances with respect to all the Yellow and Red cases in the fit data. Subsequently, parse the Test-II group into two sub-groups, and denote them as *Test-Yellow_II* and *Test-Red*. The classification rule to achieve the same is given by:

$$Class(\mathbf{x}_i) = \begin{cases} Yellow & \text{if } \dfrac{d_{Red}}{d_{Yellow}} > \dfrac{C_{YR}}{C_{RY}} \\ Red & \text{otherwise} \end{cases} \quad (5)$$

where, d_{Yellow} and d_{Red} represent the average distances to the Yellow and Red groups (fit data), respectively. The two-group model built in this step is denoted as Model_YR, as it classifies modules as either Yellow or Red. At this point all the Red modules of the test data set have been either classified correctly or misclassified.

4. Merge the Test-Yellow_I and Test-Yellow_II groups to yield the Test-Yellow group.

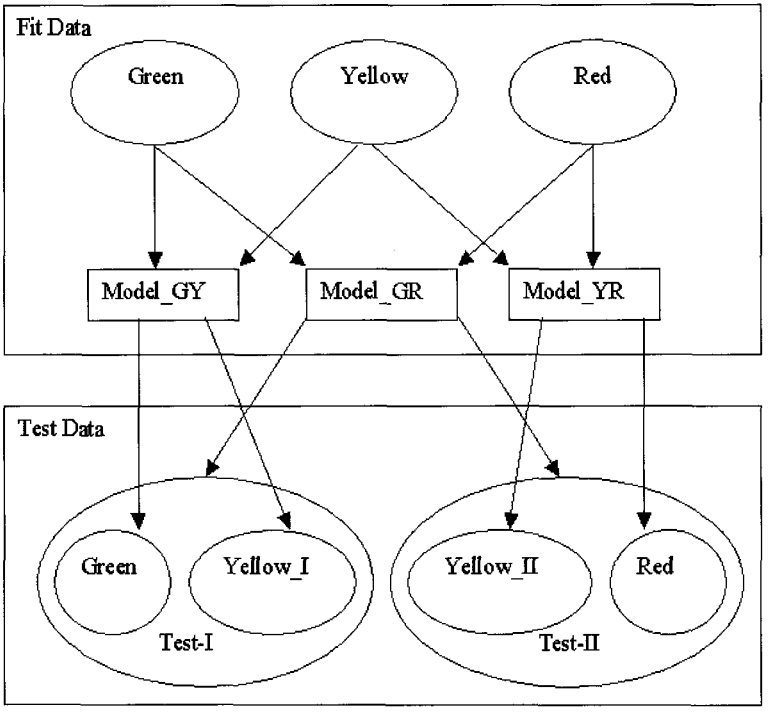


Figure 1. Three-Group Classification Algorithm

5. The above four steps are repeated for different combinations of model parameters, n_N and c. The right hand side of the inequalities in

Equations (3), (4), and (5) represent the modeling cost ratio, c, whose value can be varied to yield different three-group models. Recall, c represents the cost ratio for modeling purposes, and not the actual costs of misclassifications. The same value of modeling cost ratio, c, is used for Model_GR, Model_GY, and Model_YR, because it simplified and speeded up the automation process for the three-group model calibration. Future research will investigate the performance of the proposed algorithm in light of varying the three individual cost ratios, i.e., $\frac{C_{GR}}{C_{RG}}$, $\frac{C_{GY}}{C_{YG}}$, and $\frac{C_{YR}}{C_{RY}}$, separately.

A visual representation of the algorithm is presented in Figure 1. Model_GR, Model_GY, and Model_YR represent the models built in steps 1, 2, and 3 of the algorithm, respectively. The proposed three-group algorithm was implemented in a user-friendly empirical research tool, SMART [11], which is under development at the Empirical Software Engineering Research Laboratory, Florida Atlantic University. SMART implements CBR technology for software quality estimation, including software quality classification (two and three groups) and software fault prediction.

3. Discriminant Analysis

Discriminant analysis (DA) is a multivariate statistical technique for determining the optimum assignment (minimum misclassification) of observations into two or more classes [2,32]. Such classifications are based upon one or more quantitative measurements, such as software metrics, and it is assumed that such measurements differ between groups. In the context of this study, pure three-group classification can be obtained with DA. And since, this study evaluates the proposed algorithm with respect to a three-group model calibrated using DA, we now present a brief overview of DA.

A high degree of correlation among software measures is often observed. This is because these independent variables are usually measurements of related attributes of the given software [6]. This correlation can often lead to poorly calibrated quality estimation models.

In order to avoid the problem of multicollinearity, analysts often limit their studies to a few selected measures. However, using a large number of software measures will likely provide a more complete model.

Munson et al. introduced the application of principal components analysis (PCA) to eliminate correlation among software metrics and also reduce the number of independent variables used for model-calibration [15, 21]. In this study, principle components of software metrics are used as independent variables for the three-group classification models.

As a statistical tool, DA is available in many modern statistical packages such as the SAS and SPSS computer packages [2,32]. We used the *nonparametric* discriminant analysis approach with *stepwise* model selection. Estimated density functions for the mutually exclusive groups (high-risk, medium-risk, and low-risk) are computed using a *normal kernel* function on the vectors of the independent variables, Bayesian posterior probabilities of membership in a particular group, and the prior probability distributions of the groups in the fit data set from which the discriminant model is built [32,35].

4. Modeling Approach

Prior to calibrating software metrics-based quality estimation models, appropriate steps are needed to define the training (fit) and evaluation (test) data sets. Ideally, the test data is an independent sample of observations with respect to the fit data. Hence, if project data were available from multiple releases of a software system, such an independent sample would be possible. Unfortunately, this is rarely the case.

In cases where the data set is substantially large, an impartial data splitting technique may be appropriate to define the fit and test data sets. The data set for the case study presented was not large enough to justify data splitting. Therefore, we opted to apply the *v-fold* cross-validation technique, where v is the number of iterations used to fit and test a model.

For our case study, v is the number of modules in the fit data set, i.e., $v = 68$. At each iteration a three-group model is calibrated using $(v - 1)$

observations, and is tested with the one remaining observation. v such iterations are performed, and each iteration is tested (using the left-out singular observation) such that no two iterations use the same observation. The evaluation results are then summarized to obtain the average error rates for the trained model. Such a technique simulates the application of the model to modules of a current project with similar attributes but unknown response values.

We performed a separate principal components analysis of the observed measures in each group. This enabled us to investigate if the additional information captured by the object-oriented measures can improve a model consisting of procedural measures alone. The cumulative variance accounted for by the principle components is used as the stopping criterion when extracting components. These components are then used as inputs (independent variables) for our three-group classification problem using case-based reasoning and discriminant analysis.

4.1 Three-Group Modeling with CBR

The proposed three-group classification algorithm is illustrated by using (three-times) our previously developed two-group classification technique with CBR. The following steps summarize the modeling methodology (using SMART) adopted in calibrating and testing three-group models using the proposed technique.

1. *Case library*: The fit data set of the case study is selected as the case library. The modules in the fit data set have been marked Green, Yellow, or Red according to their number of faults.
2. *Classification technique*: The *Mahalanobis* similarity function and the generalized *Data Clustering* classification rule are selected to perform the desired classifications as illustrated in Section 2.
3. *Parameter* n_N : The number of cases to be selected from the nearest neighbor set, N, for analysis is a model parameter that is varied to yield different classification models. Values starting from 1, and as high as 15, were considered. Other higher values were investigated, but did not yield improved models.

4. *Parameter c*: The cost ratio was varied from 0.05 to 1.00, yielding different two-group classification models. Other values of c were considered, but did not yield better results.
5. *Model Fitting*: For each n_N value, the parameter c is varied. The misclassification error rates computed using the *v-fold* cross-validation technique described earlier, are summarized across the training data set.
6. *Model Selection*: The cost of the Type_RG misclassification is the most expensive from a software management point of view. Therefore, a preferred model is selected among those with *zero* (*or lowest possible*) Type_RG errors. This is because a three-group classification model should give priority in detecting all Red or high-risk modules. Furthermore, among those with the preferred Type_RG error rate, the one with the best balance between the other five error rates is selected as the final three-group model. Considering the individual costs of these five error types, a balance of equality is preferred for our case study.

4.2 Three-Group Modeling with Discriminant Analysis

Stepwise model selection procedure was performed to determine the selection of the significant independent variables, i.e., principal components of the software product measures. A significance level of 5% determined the addition and deletion of independent variables into the model selected. Using a nonparametric discriminant analysis approach [32], the smoothing parameter, λ, was varied from 0.05 to 0.15, yielding different three-group models.

4.3 Sensitivity Analysis Study

In the case of a three-group model, since we have six types of misclassification errors, we have six types of costs associated with their correction (Table 1). The costs or efforts associated with each of them, are practically speaking, not similar. The exact values for these six costs are extremely difficult to estimate at the time of modeling.

An empirical approach should be preferred, such that the different misclassification cost values considered encompass a wide spectrum of possible cost values. Therefore, a good three-group classification model should provide the minimum possible "Expected Cost of Misclassification", i.e., ECM. Furthermore, using ECM as a singular measure to perform comparative evaluation of two different classification techniques is simple and realistic, as compared to evaluating them on their misclassification rates alone.

We performed a sensitivity analysis, in which we compute the ECM values for both models, i.e., models based on the proposed three-group algorithm using CBR and discriminant analysis. The aim of such an analysis is to evaluate model-performances over a wide range of misclassification costs. Such a study gives us an insight into the stability of a classification model, because after all, the actual misclassification costs are not known until very late in the software life cycle. The ECM values for a data set with N observations are computed by,

$$ECM = \frac{1}{N}(n_{GR}C_{GR} + n_{GY}C_{GY} + n_{YR}C_{YR} + n_{YG}C_{YG} + n_{RY}C_{RY} + n_{RG}C_{RG}) \quad (6)$$

5. Case Study: A Data Communications System

5.1 System Description

The case study used for our empirical investigations of the proposed three-group classification algorithm consisted of software measurement and fault data collected from a commercial real-time data communications system, abbreviated as DC. Written in C++, the DC system contained about 7000 shipped source instructions, which represent the number of executable statements. Using the IBM Source Code Analysis and Measurement Program, SCAMP, a set of 68 software product measures were collected from each of the 68 program modules. A DC program module consisted of related source code files.

In addition to automating the collection of static software product measures to a large extent, a salient feature of SCAMP is the ability to

vary the degree of preprocessing prior to collecting measures from program modules. Therefore, it can measure modules without any, or with, full preprocessing. The latter can be performed optimally when the source code modules make substantial use of embedded macros such that they no longer conform to the grammar of the source language. However, a common side effect of full preprocessing is the inflation of many of the software product measures.

```
A_function (int arg1) {print("Greeting.\n");}
```

Figure 2. Source code example for *SSI* and *PNL*

Inflation of product measures occurs because many of the *#include* files considered by the analyzer, are building block modules of the source language. Hence, they will not depict the quality of the people developing the DC system. Practically speaking, it would not be appropriate to consider any product measures associated with building block modules when modeling the quality of a system. Therefore, for our case study we measured the DC system without any preprocessing of the program modules.

The set of 68 product metrics were categorized into two groups, i.e., procedural measures and object-oriented measures [7]. The procedural group contained 34 procedural metrics, and was applicable to both procedural and object-oriented languages. The object-oriented group consisted of 34 object-oriented metrics, and was applicable only to object-oriented languages, such as C++.

Prior to data analysis, each of the two groups was reduced by discarding measures that were either zero for all modules, non-primitive, or had a 100% correlation with another measure in the same group. A software metric that is always zero provides no software quality information, and similarly, non-primitive metrics do not provide any additional information compared to primitive measures. If two software measures were 100% correlated, then the use of PCA would yield a singular matrix. Hence, one of the measures can be discarded without any loss of information.

Table 2. Candidate Procedural Software Metrics

Measure	Description
η_1	Number of unique operators
N_1	Total number of operators
η_2	Number of unique operands
N_2	Total number of operands
$V_1(G)$	McCabe's cyclomatic number
$V_2(G)$	$V_1(G)$ plus the number of logical operators
SSI	Number of shipped source instructions. In Figure 2, SSI would be 4.
COM	Number of comments
PNL	Number of physical source lines. In Figure 2, PNL would be 1.
PNLC	Number of physical source lines containing at least one character of code
COML	Number of physical lines containing at least one character of comment
BLNL	Number of blank lines
TPPC	Number of preprocessor statements
UPPC	Unique number of preprocessor statements
AIL	Average identifier length, in characters
SM	Number of sub modules. In C++ a class, structure, union, function, or method is considered as a sub module.
SYN	Number of times SCAMP encountered a syntactical deviation

The software product measures used in our study of the DC system consisted of 17 procedural metrics (denoted as *PM*) and 20 object-oriented measures (denoted as *OOM*). These metrics are shown in Tables 2 and 3, respectively. These candidate product measures are not definitive, and are not meant for use as a guideline, or as a rule of the thumb. Other software system environments may lead to a different set of candidate software quality measures [22].

Table 3. Candidate Object Oriented Software Metrics

Measure	Description
CL	Number of classes declared
BCL	Number of base classes inherited
UBCL	Number of uniquely named base classes inherited
MEM	Total number of members declared
	In C++, a class member is any variable or function declared within a class
MEM_{PUB}	Number of members declared public
MEM_{PRI}	Number of members declared private
UMEN	Number of uniquely named members
$UMEM_{PUB}$	Number of uniquely named members declared public
$UMEM_{PRI}$	Number of uniquely named members declared private
MET	Number of methods declared
	In C++, a method is a function declared within a class
MET_{PUB}	Number of methods declared public
MET_{PRI}	Number of methods declared private
UMET	Number of uniquely named methods
$UMET_{PRI}$	Number of uniquely named methods declared private
IMET	Number of inline methods
UIMET	Number of uniquely named inline methods
VMET	Number of virtual methods
UVMET	Number of uniquely named virtual methods
OVOP	Number of overloaded operators
UOVOP	Number of uniquely named overloaded operators

The number of faults (denoted as *Faults*) was the only software process measure available for the DC system and represented the cumulative value reported by the source code control system used to

track the DC system. The value of *Faults* for a module is cumulative since its first appearance in the system.

The modules in the fit data set are labeled as follows. Modules with $Faults \leq 2$ are assigned to the low-risk or Green group, whereas those with $Faults > 19$ are assigned to the high-risk or Red group. The remaining modules, i.e., $2 < Faults \leq 19$, are categorized as medium-risk or Yellow. These threshold values were selected as per the specific needs of the DC system.

5.2 Experiments & Results

PCA performed on the 17 procedural metrics (*PM*) extracted 5 principal components that accounted for over 90% of the variance. The stopping criterion for PCA was that the Eigen values are greater than one. The first principal component encompassed the majority of the variance, whereas the fifth one accounted for the least. On the same token, PCA performed on the 20 object-oriented metrics (*OOM*) extracted 5 significant components that accounted for about 95% of the variance. As we can see the dimensionalities of the two sets of metrics is reduced considerably, and thus are now easier to work with.

Two training data sets were defined, i.e., M_P and M_{PO}. The M_P data set consisted of the 5 principal components of *PM*, whereas the M_{PO} data set, in addition to the components of *PM*, consisted of the 5 principal components of the *OOM*, for a total of 10 independent variables. For each of the three-group classification techniques, i.e., proposed algorithm (with CBR) and discriminant analysis, two models were calibrated and evaluated, and are denoted by, Model M_{PO} and Model M_P, respectively.

5.3 CBR Three-Group Models

5.3.1 Calibrating Models M_P and M_{PO}

The two models were calibrated using the modeling methodology discussed earlier. The parameters, n_N and c (cost ratio), are varied to

build different three-group classification models using CBR. We observed that for both, M_P and M_{PO} models, the parameter values of n_N = 2 and c = 0.65, yielded the preferred models. These parameters were determined empirically. In this paper we present the results of these models only.

Table 4. CBR Three-Group Model M_P
Number of Modules & Percentage

Actual	Predicted Risk Group			Total
Risk Group	Green	Yellow	Red	
Green	13	5	2	20
	65.0%	25.0%	10.0%	100.0%
Yellow	3	19	10	32
	9.4%	59.4%	31.3%	100.0%
Red	0	1	15	16
	0.0%	6.3%	93.7%	100.0%
Total	16	25	27	68
	23.5%	36.8%	39.7%	100.0%

Table 5. CBR Three-Group Model M_{PO}
Number of Modules & Percentage

Actual	Predicted Risk Group			Total
Risk Group	Green	Yellow	Red	
Green	8	9	3	20
	40.0%	45.0%	15.0%	100.0%
Yellow	3	29	0	32
	9.4%	90.6%	0.0%	100.0%
Red	0	0	16	16
	0.0%	0.0%	100.0%	100.0%
Total	11	38	19	68
	16.2%	55.9%	27.9%	100.0%

Tables 4 and 5 present the cross-validation results for Model M_P and M_{PO}, respectively. To obtain the error rates from the tables, each table can be read as a 3×3 matrix. Hence, the Type_GR and Type_RG error rates are 10.0% and 0.0%, respectively. We observe that this model detects all the high-risk modules and also has a reasonable Type_GR error rate. Similarly, the other error rates can be seen in the table.

5.3.2 *Comparing Model M_P with Model M_{PO}*

A comparative representation of the two CBR models calibrated using our proposed algorithm, is presented in Table 6. A striking observation, as seen in the table, is that no Red modules were classified as Green, by either model. Furthermore, M_{PO} identifies all Red modules correctly, indicating an improvement over the M_P model which misclassified about 6% of the Red modules as Yellow. On the same token, M_{PO} shows an improvement over M_P, when classifying the medium-risk modules.

Table 6. Three-Group Models with CBR

Misclassification	Model Type	
Error Type	M_P	M_{PO}
Type_GR	10.0%	15.0%
Type_GY	25.0%	45.0%
Type_YR	31.3%	00.0%
Type_YG	09.4%	09.4%
Type_RY	06.2%	00.0%
Type_RG	00.0%	00.0%
% Overall	30.9%	22.1%

The addition of *OOM* lowered the overall misclassification rate (by about 8%). Though it improved the classification of the Yellow modules,

the performance of M_{PO} deteriorated slightly when classifying the Green modules. Overall, an analyst may prefer the M_{PO} model, as it reduces the overall (and medium-risk) misclassification rates.

5.4 Discriminant Analysis Three-Group Models

5.4.1 Calibrating Model M_P

Stepwise discriminant analysis, selected the first four components for the model inputs, and all of these were significant at 5% [2]. A smoothing parameter value of $\lambda = 0.08$ was observed to produce the best fit. Using all the 68 observations of the data set, the quality-of-fit was perfect with no misclassifications and average uncertainty of below 0.1%. It was therefore, concluded that a three-group relationship existed between *Faults* and the procedural metrics. The *v-fold* (v = 68) cross-validation values for Model M_P are summarized in Table 7. We observe from the table, the Type_GR and Type_RG error rates are 15.0% and 6.2%, respectively.

Table 7. Discriminant Analysis Three-Group Model M_P Number of Modules & Percentage

Actual Risk Group	Predicted Risk Group			Total
	Green	Yellow	Red	
Green	13	2	5	20
	65.0%	10.0%	25.0%	100.0%
Yellow	0	23	9	32
	0.0%	71.9%	28.1%	100.0%
Red	1	4	11	16
	6.2%	25.0%	68.8%	100.0%
Total	14	29	25	68
	20.6%	42.6%	36.8%	100.0%

5.4.2 Calibrating Model M_{PO}

The model selection process selected the four principle components used above, and in addition, selected the second, third, and fourth (out of the five) principle components extracted from the *OOM*. Similar to the above model, $\lambda = 0.08$ was observed to produce the best fit. The quality-of-fit was perfect when the entire fit data was used, and hence, we concluded that a relationship existed between *Faults* and both, *PM* and *OOM*. The models misclassification rates, based on cross-validation values, are presented in Table 8.

Table 8. Discriminant Analysis Three-Group Model M_{PO} Number of Modules & Percentage

Actual Risk Group	Predicted Risk Group			Total
	Green	Yellow	Red	
Green	13	4	3	20
	65.0%	20.0%	15.0%	100.0%
Yellow	4	18	10	32
	12.5%	56.3%	31.3%	100.0%
Red	1	2	13	16
	6.2%	12.5%	81.3%	100.0%
Total	18	24	26	68
	26.5%	35.3%	38.2%	100.0%

5.4.3 Comparing Model M_P with Model M_{PO}

A comparative representation of the above two models is presented in Table 9. We observe that the overall misclassification rate of M_{PO} is

30.9%, which is lower than that of M_P. While the combined Type_GR and Type_GY errors, i.e., for the low-risk group, are the same for both models, the combined Type_RY and Type_RG error rates, i.e., for the high-risk group, of Model M_{PO} were higher than those of M_P.

Model M_{PO} depicts better classification accuracy for the medium-risk modules, i.e., combined Type_YR and Type_YG errors, as compared to Model M_P. The addition of *OOM* demonstrates improved classification of medium-risk modules, and furthermore, lowered the overall misclassification rate by about 5%.

Table 9. Discriminant Analysis Three-Group Models

Misclassification Error Type	Model Type	
	M_P	M_{PO}
Type_GR	15.0%	25.0%
Type_GY	20.0%	10.0%
Type_YR	31.3%	28.1%
Type_YG	12.5%	00.0%
Type_RY	12.5%	00.0%
Type_RG	06.2%	06.2%
% Overall	35.3%	30.9%

5.5 Discriminant Analysis vs. CBR: Three-Group Models

Among the six error types of a three-group classification model, Type_GR and Type_RG represent the extreme ends of a model's misclassification errors. The Type_GR and Type_RG errors are analogous to the Type I and Type II errors for a two-group model, indicating that cost of Type_RG is the most expensive among all the six error types. As mentioned earlier, this error should be non-existent or as low as possible.

To facilitate our discussions, we use the following notations. The four models are denoted as DA_M_P, DA_M_{PO}, CBR_M_P, and

CBR_M_{PO}, where *DA* and *CBR* indicate models built using discriminant analysis and the proposed three-group algorithm. Evaluating the misclassification errors of the M_P and M_{PO} models calibrated by the two modeling techniques (Tables 6 and 9), we observed that the proposed algorithm performed relatively better than discriminant analysis.

The overall misclassification rates for the *CBR* models, i.e., CBR_M_P (30.9%) and CBR_M_{PO} (22.1%) are lower than their respective *DA* models. This indicates a good overall performance for the proposed technique. Furthermore, except for Type_GY and Type_YR, the *CBR* models (M_P) have lower misclassification rates. This is significant because the two extreme ends of the misclassification spectrum, i.e., Type_GR and Type_RG, are preferable than those of the *DA* models.

5.5.1 *Expected Costs of Misclassifications*

The costs of the six-misclassification types can be grouped into two groups. The set of $\{C_{YR}, C_{GY}, C_{GR}\}$ formed the first group, indicating a lower risk module is identified as a higher risk module, i.e., ineffective reviews and wasted software testing efforts. The set of $\{C_{RG}, C_{RY}, C_{YG}\}$ formed the second group, indicating a higher risk module is classified as a lower risk module, i.e., missed opportunities to rectify problems prior to system operations. In the first group, from a software quality management point of view, Type_GY is the least expensive error type. On the other hand, Type_RG is the most expensive in the second group.

In our sensitivity analysis study, for each of the four models calibrated, we computed their ECM values by varying the costs of the six-misclassification types as follows. We chose three combinations for the individual costs of the first group, i.e., $\{C_{YR}, C_{GY}, C_{GR}\}$, were assigned values of {2,1,3}, {3,1,4}, and {4,1,5}. For each of these combinations, we varied the individual costs of the second group.

Practically speaking, the costs of the first group are much smaller than those of the second group. Tables 10, 11, and 12 summarize the ECM values for the DA_M_P, and CBR_M_P models. Similarly,

Tables 13, 14, and 15 summarize the ECM values for the DA_M_{PO} and CBR_M_{PO} models. The last column in the tables indicates the *relative improvement* (*RI*) of the proposed three-group classification technique over the respective *DA* models.

Table 10. ECM Values: Procedural Measures (a)

	Cost of Misclassifications						ECM		Relative
No	C_RG	C_RY	C_YG	C_YR	C_GY	C_GR	CBR	DA	Improvement
1	10	6	5	2	1	3	0.7647	1.1029	0.3066
2	10	7	5	2	1	3	0.7794	1.1324	0.3116
3	10	8	6	2	1	3	0.8382	1.2206	0.3133
4	12	7	8	2	1	3	0.9118	1.3382	0.3187
5	12	8	6	2	1	3	0.8382	1.25	0.3294
6	12	10	6	2	1	3	0.8676	1.3088	0.3371
7	15	7	5	2	1	3	0.7794	1.2059	0.3537
8	15	8	6	2	1	3	0.8382	1.2941	0.3523
9	15	10	8	2	1	3	0.9559	1.4706	0.3500
10	20	10	6	2	1	3	0.8676	1.4265	0.3917
11	20	12	8	2	1	3	0.9853	1.6029	0.3853
12	20	15	10	2	1	3	1.1176	1.8088	0.3821
13	25	15	8	2	1	3	1.0294	1.7647	0.4167
14	25	15	10	2	1	3	1.1176	1.8824	0.4062
15	30	10	5	2	1	3	0.8235	1.5147	0.4563
16	30	15	5	2	1	3	0.8971	1.6618	0.4602
17	30	15	10	2	1	3	1.1176	1.9559	0.4285
18	30	20	15	2	1	3	1.4118	2.3971	0.4110
19	40	25	15	2	1	3	1.4853	2.6912	0.4481
20	40	30	10	2	1	3	1.3382	2.5441	0.4740
21	50	25	10	2	1	3	1.2647	2.5441	0.5029
22	50	30	15	2	1	3	1.5588	2.9853	0.4778
23	80	20	15	2	1	3	1.4118	3.1324	0.5493
24	80	40	20	2	1	3	1.9265	4.0147	0.5201
25	80	50	20	2	1	3	2.0735	4.3088	0.5188
26	100	25	10	2	1	3	1.2647	3.2794	0.6144
27	100	50	25	2	1	3	2.2941	4.8971	0.5315

Table 10. ECM Values: Procedural Measures (a) (cont.)

No	Cost of Misclassifications						ECM		Relative Improvement
	C_RG	C_RY	C_YG	C_YR	C_GY	C_GR	CBR	DA	
28	100	60	40	2	1	3	3.1029	6.0735	0.4891
29	100	75	50	2	1	3	3.7647	7.1029	0.4700
30	120	40	20	2	1	3	1.9265	4.6029	0.5815
31	120	70	50	2	1	3	3.6912	7.25	0.4909
32	150	25	10	2	1	3	1.2647	4.0147	0.6850
33	150	50	10	2	1	3	1.6324	4.75	0.6563
34	150	80	50	2	1	3	3.8382	7.9853	0.5193
35	150	100	10	2	1	3	2.3676	6.2206	0.6194
36	150	100	20	2	1	3	2.8088	6.8088	0.5875
37	150	100	50	2	1	3	4.1324	8.5735	0.5180
38	200	50	15	2	1	3	1.8529	5.7794	0.6794
39	200	70	30	2	1	3	2.8088	7.25	0.6126
40	200	90	50	2	1	3	3.9853	9.0147	0.5579
41	200	100	80	2	1	3	5.4559	11.0735	0.5073
42	200	120	100	2	1	3	6.6324	12.8382	0.4834
43	200	150	50	2	1	3	4.8676	10.7794	0.5484
44	200	150	100	2	1	3	7.0735	13.7206	0.4845

Table 11. ECM Values: Procedural Measures (b)

No	Cost of Misclassifications						ECM		Relative Improvement
	C_RG	C_RY	C_YG	C_YR	C_GY	C_GR	CBR	DA	
	0						.9412	.2941	.2727
	0						.9559	.3235	.2777
	0						.0147	.4118	.2813
	2						.9559	.3529	.2935
	2						.0147	.4412	.2959
	2	0					.0441	.5	.3039
	5						.9559	.3971	.3158
	5						.0147	.4853	.3168
	5	0					.1324	.6618	.3186
0	0	0					.0441	.6176	.3545

Table 11. ECM Values: Procedural Measures (b) (cont.)

	Cost of Misclassifications						ECM		Relative
No	C_RG	C_RY	C_YG	C_YR	C_GY	C_GR	CBR	DA	Improvement
1	0	2					.1618	.7941	.3525
2	0	5	0				.2941		.3530
3	5	5					.2059	.9559	.3835
4	5	5	0				.2941	.0735	.3759
5	0	0						.7059	.4138
6	0	5					.0735	.8529	.4206
7	0	5	0				.2941	.1471	.3972
8	0	0	5				.5882	.5882	.3864
9	0	5	5				.6618	.8824	.4235
0	0	0	0				.5147	.7353	.4462
1	0	5	0				.4412	.7353	.4731
2	0	0	5				.7353	.1765	.4537
3	0	0	5				.5882	.3235	.5221
4	0	0	0				.1029	.2059	.5000
5	0	0	0				.25	.5	.5000
6	00	5	0				.4412	.4706	.5847
7	00	0	5				.4706	.0882	.5144
8	00	0	0				.2794	.2647	.4765
9	00	5	0				.9412	.2941	.4597
0	20	0	0				.1029	.7941	.5614
1	20	0	0				.8676	.4412	.4802
2	50	5	0				.4412	.2059	.6573
3	50	0	0				.8088	.9412	.6339
4	50	0	0				.0147	.1765	.5090
5	50	00	0				.5441	.4118	.6032
6	50	00	0				.9853		.5735
7	50	00	0				.3088	.7647	.5084
8	00	0	5				.0294	.9706	.6601
9	00	0	0				.9853	.4412	.5988
0	00	0	0				.1618	.2059	.5479
1	00	00	0				.6324	1.2647	.5000
2	00	20	00				.8088	3.0294	.4774
3	00	50	0				.0441	0.9706	.5402
4	00	50	00				.25	3.9118	.4789

In the case of the M_P models (Tables 10, 11, and 12), the *CBR* models reported at least 24% lower ECM values. Furthermore, as the costs are increased, the performance of the *DA* model deteriorates drastically, as seen by the improved *RI* values. This suggests that the proposed

technique is relatively less sensitive, i.e., more stable, to the variation of misclassification costs.

Table 12. ECM Values - Procedural Measures (c)

	Cost of Misclassifications						ECM		Relative
No	C_RG	C_RY	C_YG	C_YR	C_GY	C_GR	CBR	DA	Improvement
1	10	6	5	4	1	5	1.1176	1.4853	0.2475
2	10	7	5	4	1	5	1.1324	1.5147	0.2525
3	10	8	6	4	1	5	1.1912	1.6029	0.2569
4	12	7	5	4	1	5	1.1324	1.5441	0.2667
5	12	8	6	4	1	5	1.1912	1.6324	0.2703
6	12	10	6	4	1	5	1.2206	1.6912	0.2783
7	15	7	5	4	1	5	1.1324	1.5882	0.2871
8	15	8	6	4	1	5	1.1912	1.6765	0.2895
9	15	10	8	4	1	5	1.3088	1.8529	0.2936
10	20	10	6	4	1	5	1.2206	1.8088	0.3252
11	20	12	6	4	1	5	1.25	1.8676	0.3307
12	20	15	10	4	1	5	1.4706	2.1912	0.3289
13	25	15	8	4	1	5	1.3824	2.1471	0.3562
14	25	15	10	4	1	5	1.4706	2.2647	0.3506
15	30	10	5	4	1	5	1.1765	1.8971	0.3798
16	30	15	5	4	1	5	1.25	2.0441	0.3885
17	30	15	10	4	1	5	1.4706	2.3382	0.3711
18	30	20	15	4	1	5	1.7647	2.7794	0.3651
19	40	25	15	4	1	5	1.8382	3.0735	0.4019
20	40	30	10	4	1	5	1.6912	2.9265	0.4221
21	50	25	10	4	1	5	1.6176	2.9265	0.4472
22	50	30	15	4	1	5	1.9118	3.3676	0.4323
23	80	20	15	4	1	5	1.7647	3.5147	0.4979
24	80	40	20	4	1	5	2.2794	4.3971	0.4816
25	80	50	20	4	1	5	2.4265	4.6912	0.4828
26	100	25	10	4	1	5	1.6176	3.6618	0.5582
27	100	50	25	4	1	5	2.6471	5.2794	0.4986
28	100	60	40	4	1	5	3.4559	6.4559	0.4647
29	100	75	50	4	1	5	4.1176	7.4853	0.4499
30	120	40	20	4	1	5	2.2794	4.9853	0.5428
31	120	70	50	4	1	5	4.0441	7.6324	0.4701
32	150	25	10	4	1	5	1.6176	4.3971	0.6321
33	150	50	10	4	1	5	1.9853	5.1324	0.6132
34	150	80	50	4	1	5	4.1912	8.3676	0.4991
35	150	100	10	4	1	5	2.7206	6.6029	0.5880

Table 12. ECM Values - Procedural Measures (c) (cont.)

	Cost of Misclassifications						ECM		Relative
No	C_RG	C_RY	C_YG	C_YR	C_GY	C_GR	CBR	DA	Improvement
36	150	100	20	4	1	5	3.1618	7.1912	0.5603
37	150	100	50	4	1	5	4.4853	8.9559	0.4992
38	200	50	15	4	1	5	2.2059	6.1618	0.6420
39	200	70	30	4	1	5	3.1618	7.6324	0.5857
40	200	90	50	4	1	5	4.3382	9.3971	0.5383
41	200	100	80	4	1	5	5.8088	11.4559	0.4929
42	200	120	100	4	1	5	6.9853	13.2206	0.4716
43	200	150	50	4	1	5	5.2206	11.1618	0.5323
44	200	150	100	4	1	5	7.4265	14.1029	0.4734

Table 13. ECM Values - Procedural and Object-Oriented Measures (a)

	Cost of Misclassifications						ECM		Relative
No	C_RG	C_RY	C_YG	C_YR	C_GY	C_GR	CBR	DA	Improvement
1	10	6	5	2	1	3	0.4853	1.0147	0.5217
2	10	7	5	2	1	3	0.4853	1.0735	0.5479
3	10	8	6	2	1	3	0.5294	1.1324	0.5324
4	12	7	8	2	1	3	0.6176	1.1029	0.4400
5	12	8	6	2	1	3	0.5294	1.1618	0.5443
6	12	10	6	2	1	3	0.5294	1.2794	0.5862
7	15	7	5	2	1	3	0.4853	1.1471	0.5769
8	15	8	6	2	1	3	0.5294	1.2059	0.5610
9	15	10	8	2	1	3	0.6176	1.3235	0.5334
10	20	10	6	2	1	3	0.5294	1.3971	0.6210
11	20	12	8	2	1	3	0.6176	1.5147	0.5923
12	20	15	10	2	1	3	0.7059	1.6912	0.5826
13	25	15	8	2	1	3	0.6176	1.7647	0.6500
14	25	15	10	2	1	3	0.7059	1.7647	0.6000
15	30	10	5	2	1	3	0.4853	1.5441	0.6857
16	30	15	5	2	1	3	0.4853	1.8382	0.7360
17	30	15	10	2	1	3	0.7059	1.8382	0.6160
18	30	20	15	2	1	3	0.9265	2.1324	0.5655
19	40	25	15	2	1	3	0.9265	2.5735	0.6400
20	40	30	10	2	1	3	0.7059	2.8676	0.7539
21	50	25	10	2	1	3	0.7059	2.7206	0.7405
22	50	30	15	2	1	3	0.9265	3.0147	0.6927
22	50	30	15	2	1	3	0.9265	3.0147	0.6927
23	80	20	15	2	1	3	0.9265	2.8676	0.6769

Table 13. ECM Values - Procedural and Object-Oriented Measures (a) (cont.)

No	Cost of Misclassifications						ECM		Relative Improvement
	C_RG	C_RY	C_YG	C_YR	C_GY	C_GR	CBR	DA	
24	80	40	20	2	1	3	1.1471	4.0441	0.7164
25	80	50	20	2	1	3	1.1471	4.6324	0.7524
26	100	25	10	2	1	3	0.7059	3.4559	0.7957
27	100	50	25	2	1	3	1.3676	4.9265	0.7224
28	100	60	40	2	1	3	2.0294	5.5147	0.6320
29	100	75	50	2	1	3	2.4706	6.3971	0.6138
30	120	40	20	2	1	3	1.1471	4.6324	0.7524
31	120	70	50	2	1	3	2.4706	6.3971	0.6138
32	150	25	10	2	1	3	0.7059	4.1912	0.8316
33	150	50	10	2	1	3	0.7059	5.6618	0.8753
34	150	80	50	2	1	3	2.4706	7.4265	0.6673
35	150	100	10	2	1	3	0.7059	8.6029	0.9180
36	150	100	20	2	1	3	1.1471	8.6029	0.8667
37	150	100	50	2	1	3	2.4706	8.6029	0.7128
38	200	50	15	2	1	3	0.9265	6.3971	0.8552
39	200	70	30	2	1	3	1.5882	7.5735	0.7903
40	200	90	50	2	1	3	2.4706	8.75	0.7176
41	200	100	80	2	1	3	3.7941	9.3382	0.5937
42	200	120	100	2	1	3	4.6765	10.5147	0.5552
43	200	150	50	2	1	3	2.4706	12.2794	0.7988
44	200	150	100	2	1	3	4.6765	12.2794	0.6192

Table 14. ECM Values – Procedural and Object-Oriented Measures (b)

No	Cost of Misclassifications						ECM		Relative Improvement
	C_RG	C_RY	C_YG	C_YR	C_GY	C_GR	CBR	DA	
1	10	6	5	3	1	4	0.5294	1.2206	0.5663
2	10	7	5	3	1	4	0.5294	1.2794	0.5862
3	10	8	6	3	1	4	0.5735	1.3382	0.5714
4	12	7	5	3	1	4	0.5294	1.3088	0.5955
5	12	8	6	3	1	4	0.5735	1.3676	0.5807
6	12	10	6	3	1	4	0.5735	1.4853	0.6139
7	15	7	5	3	1	4	0.5294	1.3529	0.6087
8	15	8	6	3	1	4	0.5735	1.4118	0.5937
9	15	10	8	3	1	4	0.6618	1.5294	0.5673
10	20	10	6	3	1	4	0.5735	1.6029	0.6422
11	20	12	8	3	1	4	0.6618	1.7206	0.6154
12	20	15	10	3	1	4	0.75	1.8971	0.6047
13	25	15	8	3	1	4	0.6618	1.9706	0.6642
14	25	15	10	3	1	4	0.75	1.9706	0.6194
15	30	10	5	3	1	4	0.5294	1.75	0.6975

Table 14. ECM Values - Procedural and Object-Oriented Measures (b) (cont.)

	Cost of Misclassifications						ECM		Relative
No	C_RG	C_RY	C_YG	C_YR	C_GY	C_GR	CBR	DA	Improvement
16	30	15	5	3	1	4	0.5294	2.0441	0.7410
17	30	15	10	3	1	4	0.75	2.0441	0.6331
18	30	20	15	3	1	4	0.9706	2.3382	0.5849
19	40	25	15	3	1	4	0.9706	2.7794	0.6508
20	40	30	10	3	1	4	0.75	3.0735	0.7560
21	50	25	10	3	1	4	0.75	2.9265	0.7437
22	50	30	15	3	1	4	0.9706	3.2206	0.6986
23	80	20	15	3	1	4	0.9706	3.0735	0.6842
24	80	40	20	3	1	4	1.1912	4.25	0.7197
25	80	50	20	3	1	4	1.1912	4.8382	0.7538
26	100	25	10	3	1	4	0.75	3.6618	0.7952
27	100	50	25	3	1	4	1.4118	5.1324	0.7249
28	100	60	40	3	1	4	2.0735	5.7206	0.6375
29	100	75	50	3	1	4	2.5147	6.6029	0.6192
30	120	40	20	3	1	4	1.1912	4.8382	0.7538
31	120	70	50	3	1	4	2.5147	6.6029	0.6192
32	150	25	10	3	1	4	0.75	4.3971	0.8294
33	150	50	10	3	1	4	0.75	5.8676	0.8722
34	150	80	50	3	1	4	2.5147	7.6324	0.6705
35	150	100	10	3	1	4	0.75	8.8088	0.9149
36	150	100	20	3	1	4	1.1912	8.8088	0.8648
37	150	100	50	3	1	4	2.5147	8.8088	0.7145
38	200	50	15	3	1	4	0.9706	6.6029	0.8530
39	200	70	30	3	1	4	1.6324	7.7794	0.7902
40	200	90	50	3	1	4	2.5147	8.9559	0.7192
41	200	100	80	3	1	4	3.8382	9.5441	0.5978
42	200	120	100	3	1	4	4.7206	10.7206	0.5597
43	200	150	50	3	1	4	2.5147	12.4853	0.7986
44	200	150	100	3	1	4	4.7206	12.4853	0.6219

Evaluating the two M_{PO} models (Tables 13, 14, and 15), once again we observe that *CBR* models yielded better ECM values, i.e., at least about 52% better. The *RI* values increased as the costs are increased, suggesting the stability of the proposed technique with respect to *DA*. Furthermore, the addition of the *OOM* yielded better ECM values for both techniques. This would indicate that object-oriented measures could

be used to improve classification of models based on procedural measures.

Table 15. ECM Values - Procedural and Object-Oriented Measures (c)

	Cost of Misclassifications						ECM		Relative
No	C_RG	C_RY	C_YG	C_YR	C_GY	C_GR	CBR	DA	Improvement
1	10	6	5	4	1	5	0.5735	1.4265	0.5979
2	10	7	5	4	1	5	0.5735	1.4853	0.6139
3	10	8	6	4	1	5	0.6176	1.5441	0.6000
4	12	7	5	4	1	5	0.5735	1.5147	0.6214
5	12	8	6	4	1	5	0.6176	1.5735	0.6075
6	12	10	6	4	1	5	0.6176	1.6912	0.6348
7	15	7	5	4	1	5	0.5735	1.5588	0.6321
8	15	8	6	4	1	5	0.6176	1.6176	0.6182
9	15	10	8	4	1	5	0.7059	1.7353	0.5932
10	20	10	6	4	1	5	0.6176	1.8088	0.6586
11	20	12	6	4	1	5	0.6176	1.9265	0.6794
12	20	15	10	4	1	5	0.7941	2.1029	0.6224
13	25	15	8	4	1	5	0.7059	2.1765	0.6757
14	25	15	10	4	1	5	0.7941	2.1765	0.6351
15	30	10	5	4	1	5	0.5735	1.9559	0.7068
16	30	15	5	4	1	5	0.5735	2.25	0.7451
17	30	15	10	4	1	5	0.7941	2.25	0.6471
18	30	20	15	4	1	5	1.0147	2.5441	0.6012
19	40	25	15	4	1	5	1.0147	2.9853	0.6601
20	40	30	10	4	1	5	0.7941	3.2794	0.7579
21	50	25	10	4	1	5	0.7941	3.1324	0.7465
22	50	30	15	4	1	5	1.0147	3.4265	0.7039
23	80	20	15	4	1	5	1.0147	3.2794	0.6906
24	80	40	20	4	1	5	1.2353	4.4559	0.7228
25	80	50	20	4	1	5	1.2353	5.0441	0.7551
26	100	25	10	4	1	5	0.7941	3.8676	0.7947
27	100	50	25	4	1	5	1.4559	5.3382	0.7273
28	100	60	40	4	1	5	2.1176	5.9265	0.6427
29	100	75	50	4	1	5	2.5588	6.8088	0.6242
30	120	40	20	4	1	5	1.2353	5.0441	0.7551
31	120	70	50	4	1	5	2.5588	6.8088	0.6242
32	150	25	10	4	1	5	0.7941	4.6029	0.8275
33	150	50	10	4	1	5	0.7941	6.0735	0.8693
34	150	80	50	4	1	5	2.5588	7.8382	0.6735
35	150	100	10	4	1	5	0.7941	9.0147	0.9119
36	150	100	20	4	1	5	1.2353	9.0147	0.8630
37	150	100	50	4	1	5	2.5588	9.0147	0.7162
38	200	50	15	4	1	5	1.0147	6.8088	0.8510
39	200	70	30	4	1	5	1.6765	7.9853	0.7901

Table 15. ECM Values - Procedural and Object-Oriented Measures (c) (cont.)

	Cost of Misclassifications						ECM		Relative
No	C_RG	C_RY	C_YG	C_YR	C_GY	C_GR	CBR	DA	Improvement
40	200	90	50	4	1	5	2.5588	9.1618	0.7207
41	200	100	80	4	1	5	3.8824	9.75	0.6018
42	200	120	100	4	1	5	4.7647	10.9265	0.5639
43	200	150	50	4	1	5	2.5588	12.6912	0.7984
44	200	150	100	4	1	5	4.7647	12.6912	0.6246

6. Summary

A two-group software quality classification (SQC) model assumes that enough quality enhancements and software testing resources are available to all modules identified as fault-prone. However, project resources, especially for software testing and verification, are almost always limited and finite. Such a restrictive scenario may lead to some (or many) risk-prone modules to remain untested or unverified.

A three-group SQC model may be more effective in segregating the high-risk modules, facilitating a more focussed and achievable software testing and quality improvement endeavor. For example, all modules predicted as Green will not be inspected, whereas all those estimated as Red will be subjected to further software testing and quality improvement efforts. The modules predicted as medium-risk might, depending on resource availability, be reviewed using only a fraction (about 15% or 20%) of time and effort used for the Red modules. Furthermore, a three-group classification model may also be desired when the risk factor, such as number of faults, varies considerably with many discrete values. However, the computational complexity of developing a three-group technique may be much higher than that of the more commonly used two-group technique.

This study introduces an innovative three-group classification technique that circumvents the need for developing a direct method to classify observations into three groups, such as high-risk, medium-risk, and low-risk. The proposed technique follows a very simple and logical approach, by utilizing a two-group classification technique three times. The combinations of these three iterations, classifies modules into any of

the three groups. A striking feature of the technique is that it can be applied to any existing two-group SQC modeling technique, such as neural networks, genetic programming, and logistic regression.

The six-misclassification errors of a three-group model are associated with their respective corrective costs and effort needed to rectify the error. Software management would agree that the misclassification costs are not similar, because some involve ineffective software testing efforts, whereas others include corrective efforts on faults discovered during operations and post-deployment. Therefore, when evaluating performance of such models, the use of error rates alone is not realistic.

Our previously developed analogy-based (CBR) two-group SQC technique is utilized to demonstrate an empirical application of the proposed algorithm. The performance of the calibrated three-group model is compared against a three-group nonparametric discriminant model. We used a unified singular measure, i.e., expected cost of misclassification, to evaluate the model-performance of a three-group model. In addition, the measure was also used in our comparative evaluation of the three-group modeling techniques, i.e., proposed algorithm with CBR and discriminant analysis.

The case study consisted of procedural and object-oriented software product measures extracted from a commercial real-time data communications system, written in C++. Two training data sets were defined, and included principal components of the object-oriented and/or procedural software measures. The first data set contained principal components of the procedural metrics, whereas the second data set contained principal components of both, procedural and object-oriented measures.

Our overall observation concluded that the proposed three-group SQC modeling technique using CBR demonstrated promising results as compared to a three-group discriminant analysis model. The CBR-based model demonstrated lower ECM values, and in addition, when compared with the discriminant model, the sensitivity of the model with respect to varying misclassification costs was very low. The stability of the model is advantageous, because the actual costs of misclassifications are unknown at the time of modeling.

It was also observed that the addition of object-oriented software product measures improved the overall classification accuracy of a model based on procedural measures alone. More interestingly, they improved the classification of medium-risk modules dramatically. An analyst of the data communication system may prefer to use a model that includes object-oriented measures, since the overall misclassification rate is reduced.

Our overall observation regarding the proposed three-group classification algorithm, in terms of good classification accuracy and model-stability, shows promising results. In addition, the simplicity and automation (via SMART) of the proposed algorithm makes it a good choice for other three-group classification problems. This study is our first attempt at illustrating the proposed algorithm. However, significant research in the near future will further validate the usefulness of the proposed three-group classification algorithm as an aid for effective software testing and reliability enhancements.

Future work will investigate the performance of the proposed modeling algorithm with case studies of other industrial software systems. Instead of using a common cost ratio, c, for the three two-group models used by the proposed algorithm, investigative empirical research will be performed by varying their individual cost ratios. In addition, the proposed algorithm will be investigated with other underlying two-group classification techniques, such as logistic regression and neural networks.

Acknowledgements

We thank the anonymous reviewer for providing useful suggestions and comments. We also thank: Robert M. Szabo for his contributions with data collection and analysis; Huiming Song for his assistance in the development of SMART; Kehan Gao for her computational assistance; and Erik Geleyn and Laurent Nguyen for their editorial help. This work was supported in part by: Cooperative Agreement NC 2-1141 from NASA Ames Research Center, Software Technology Division; Center Software Initiative for the NASA Software IV&V Facility at Fairmont, West Virginia; and National Science Foundation Grant CCR-9970893.

References

[1] B. Beizer. *Software Testing Techniques*. ITP: Van Nostrand Rienhold, New York, NY USA, Second edition, 1990.

[2] M. L. Berenson, D. M. Levine, and M. Goldstein. *Intermediate Statistical Methods and Applications: A Computer Package Approach*. Prentice Hall, Englewood Cliffs, NJ, USA, 1983.

[3] L. C. Briand and V. R. Basili. "A classification procedure for the effective management of changes during the maintenance process". In *Proceedings: International Conference on Software Maintenance*, pages 328-336, Orlando, Florida USA, November 1992. IEEE Computer Society.

[4] L. C. Briand, V. R. Basili, and C. J. Hetmanski. "Developing interpretable models with optimized set reduction for identifying high-risk software components". *IEEE Transactions on Software Engineering*, 19(11):1028-1044, Nov. 1993.

[5] L. C. Briand, W. L. Melo, and J. Wust. "Assessing the applicability of fault-proneness models across object-oriented software projects". *IEEE Transactions on Software Engineering*, 28(7):706-720, July 2002.

[6] C. Ebert. "Classification techniques for metric-based software development". *Software Quality Journal*, 5(4):255-272, Dec. 1996.

[7] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Company: ITP, Boston, MA USA, second edition, 1997.

[8] T. M. Khoshgoftaar and E. B. Allen. "Logistic regression modeling of software quality". *International Journal of Reliability, Quality and Safety Engineering*, 6(4):303-317, Dec. 1999.

[9] T. M. Khoshgoftaar and E. B. Allen. "A practical classification rule for software quality models". *IEEE Transactions on Reliability*, 49(2):209-216, June 2000.

[10] T. M. Khoshgoftaar and E. B. Allen. "Modeling software quality with classification trees". In H. Pham, editor, *Recent Advances in Reliability and Quality Engineering*, chapter 15, pages 247-270. World Scientific Publishing, Singapore, 2001.

[11] T. M. Khoshgoftaar, E. B. Allen, and J. C. Busboom. "Modeling software quality: The software measurement analysis and reliability toolkit". *In Proceedings: International Conference on Tools with Artificial Intelligence*, pages 54-61, Nov 2000.

[12] T. M. Khoshgoftaar, E. B. Allen, and J. Deng. "Controlling overfitting in software quality models: Experiments with regression trees and classification". *In Proceedings: Seventh International Software Metrics Symposium*, pages 190-198, London UK, April 2001. IEEE Computer Society.

[13] T. M. Khoshgoftaar, E. B. Allen, and R. Shan. "Improving tree-based models of software quality with principal components analysis". In *Proceedings or the Eleventh International Symposium on Software Reliability Engineering*, pages 198-209, San Jose, California USA, Oct. 2000. IEEE Computer Society.

[14] T. M. Khoshgoftaar, B. Cukic, and N. Seliya. "Predicting fault-prone modules in embedded systems using analogy based classification models". *International*

Journal of Software Engineering and Knowledge Engineering, 12(2):201-221, April 2002.

[15] T. M. Khoshgoftaar and D. L. Lanning. "A neural network approach for early detection of program modules having high risk in the maintenance phase". *Journal of Systems and Software*, 29(1):85-91, Apr. 1995.

[16] T. M. Khoshgoftaar, X. Yuan, and D. L. Lanning. "Balancing misclassification rates in classification tree models of software quality". *Empirical Software Engineering*, 5:313-330, 2000.

[17] J. Kolodner. *Case-Based Reasoning*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.

[18] Y. Liu and T. M. Khoshgoftaar. "Genetic programming model for software quality prediction". In *Proceedings of Sixth IEEE International High Assurance Systems Engineering Symposium*, pages 127-136, Boca Raton, Florida USA, October 2001. IEEE Computer Society.

[19] J. C. Munson and T. M. Khoshgoftaar. "Software metrics for reliability assessment". In M. Lyu, editor, *Handbook of Software Reliability Engineering*, chapter 12, pages 493-529. McGraw-Hill, New York, 1996.

[20] N. Ohlsson and H. Alberg. "Predicting fault-prone software modules in telephone switches". *IEEE Transactions on Software Engineering*, 22 (12): 886-894, 1996.

[21] M. C. Ohlsson, M. Helander, and C. Wohlin. "Quality Improvement by Identification of Fault-Prone Modules using Software Design Metrics". In *Proceedings: International Conference on Software Quality*, pages 1-13, Ottawa, Canada, 1996.

[22] M. C. Ohlsson and P. Runeson. "Experience from Replicating Empirical Studies on Prediction Models". In *Proceedings: 8th International Software Metrics Symposium*, pages 217-226, Ottawa, Canada, June 2002. IEEE Computer Society.

[23] Y. Ping, T. Systa, and H. Muller. "Predicting fault-proneness using OO metrics: An industrial case study". In, Gyimothy and F. B. Abreu, editors, *Proceedings: 6th European Conference on Software Maintenance and Reengineering*, pages 99-107, Budapest, Hungary, March 2002.

[24] N. J. Pizzi, A. R. Summers, and W. Pedrycz. "Software quality prediction using median-adjusted class labels". In *Proceedings: International Joint Conference on Neural Networks*. Vol. 3, pages 2405-2409, Honolulu, Hawaii, USA, May 2002. IEEE Computer Society.

[25] C. V. Ramamoorthy, C. Chandra, S. Ishihara, and Y. Ng. "Knowledge based tools for risk assessment in software development and reuse". In *Proceedings: Sixth International Conference on Tools with Artificial Intelligence*, pages 364-371. IEEE Computer Society, 1993.

[26] F. D. Ross. *An empirical study of analogy based software quality classification models*. Master's thesis, Florida Atlantic University, Boca Raton, FL USA, August 2001. Advised by T. M. Khoshgoftaar.

[27] P. Runeson, M. C. Ohlsson, and C. Wohlin. *A Classification Scheme for Studies on Fault-Prone Components*. Lecture Notes in Computer Science, vol. 2188, pp. 341-355, 2001. Springer Link.

[28] N. F. Scheidewind. "Validating software metrics: Producing quality discriminators". In *Proceedings: International Symposium on Software Reliability Engineering*, pages 225-232, Austin, Texas USA, May 1991. IEEE Computer Society.

[29] N. F. Scheidewind. "Methodology for validating metrics". *IEEE Transactions on Software Engineering*, 18(5):410-422, August 1992.

[30] N. F. Schneidewind. "Investigation of logistic regression as a discriminant of software quality". In *Proceedings: 7th International Software Metrics Symposium*, pages 328-337, London, UK, April 2001. IEEE Computer Society.

[31] N. F. *Schneidewind. Body of Knowledge for Software Quality Measurement*, IEEE Computer, 35(2)77-83, February 2002.

[32] G. A. F. Seber. *Multivariate Observations*. John Wiley and Sons, New York, NY USA, 1984.

[33] M. Shepperd and C. Schofield. "Estimating software project effort using analogies". *IEEE Transactions on Software Engineering*, 23(12):736-743, November 1997.

[34] A. Suarez and J. F. Lutsko. "Globally Optimal Fuzzy Decision Trees for Classification and Regression". *Pattern Analysis and Machine Intelligence*, 21(12):1297-1311, 1999. IEEE Computer Society.

[35] R. M. Szabo and T. M. Khoshgoftaar. "Classifying software modules into three risk groups". In H. Pham and M.-W. Lu, editors, *Proceedings: Sixth International Conference on Reliability and Quality in Design*, pages 90-95, Orlando, Florida USA, August 1999. International Society of Science and Applied Technologies.

[36] S. Wang and D. Kountanis. "IASCE: An intelligent assistant to software cost estimation". In *Proceedings: Fifth International Conference on Tools with Artificial Intelligence*, pages 114-176, Arlington, VA USA, 1992. IEEE Computer Society.

CHAPTER 6

DATA MINING WITH RESAMPLING IN SOFTWARE METRICS DATABASES

Scott Dick

University of Alberta
Department of Electrical and Computer Engineering
Edmonton, AB, Canada
E-mail: dick@ee.ualberta.ca

Abraham Kandel

University of South Florida
Department of Computer Science & Engineering
Tampa, FL, USA
E-mail: kandel@csee.usf.edu

We consider the well-known problem of skewness in publicly available datasets of software metrics. In general, most modules in a software system are usually small, and not prone to failure. Only a few of these modules will be large, complex, and prone to failure. Those few modules, however, are precisely the modules that a developer is interested in identifying. This skewness distorts the results of both statistical analysis and machine learning, rendering the results of virtually any data mining algorithm suspect. We will, for the first time, use the machine learning technique of *resampling* a dataset to overcome the problem of skewness in software metrics databases. We investigate the use of resampling in three datasets of software metrics, and how resampling alters the results of a data mining algorithm. We find that resampling can calibrate a C4.5 decision tree generator to identify the few risky modules in a software system very accurately, as measured by the geometric mean testing accuracy in those classes. A discussion of how these results might be used in a practical setting follows our experimental results.

1. Introduction

In the words of Frederick Brooks, software is "pure thought-stuff" [7]. Software seems so easy to create, and so easy to change; a developer just types some symbols on a keyboard, and a product is created or modified. At least, so it seems. The difficulty in modern software systems is that they are so incredibly huge that one person cannot possibly comprehend the complete system -- or understand how any changes they make might affect other, untouched parts of the software. This fact has resulted in a number of high-profile software disasters in the last ten years, not to mention the more mundane loss of huge sums of money. Software failures increasingly put life, limb and property at risk. The USA Department of Defense recently disclosed that they are losing over four *billion* dollars a year due to faulty software. Software quality is, quite simply, the most difficult and important technological challenge of the 21st century.

Software developers are constantly trying to improve the quality of the software they develop. This is done in two ways: by extensive testing, and by trying to improve their development processes. One of the key elements in the latter effort is to use software metrics to measure the complexity of software modules. Three decades of research has shown that there is apparently a linear relationship between the failure rate of a module and the complexity of that module [34], [35]. This relationship holds both for simple metrics like the number of lines of code, and for structural metrics like McCabe's cyclomatic complexity [37]. With metric values available to them, developers can attempt to use basic quality-control techniques. Typically, modules will be ranked in order of increasing complexity, and additional resources will be directed at those modules having the highest complexity [30].

While there appears to be an overall linear relationship between metric values and failure rates, this relationship is not fully understood. At this point, we can say that a module with a McCabe's complexity of 20 is much more likely to fail than one with a McCabe's complexity of 5; however, we cannot quantify this assertion. No theory exists to describe the proper model form to relate software metrics and failure rates; hence, there is no way to use parametric models such as multiple regression.

The space of possible model forms is uncountably infinite, and there is no guidance available on selecting any one model form over another. Furthermore, the use of multiple metrics is complicated by the fact that metrics are linearly related to *each other*, as well as to the failure rate. This phenomenon of *multicollinearity* renders statistical regression useless, since independence among regressor variables is a fundamental assumption of the regression algorithms. Machine learning algorithms, on the other hand, are able to operate in the presence of multicollinearity, as demonstrated in [30], [1], [10], [18] and [27], among others. However, machine learning suffers from a different problem: databases of metrics for any project are heavily skewed towards modules with low metric values and low failure rates. Skewness distorts a machine learning algorithm because the algorithm is attempting to optimize a global performance index. For instance, if a particular dataset contains 95% small, safe modules and 5% risky models, a machine learner can simply guess that *any* module is small and safe -- and thereby achieve an accuracy of 95%, which is considered very high [9]. Since the data themselves are skewed, the model a machine learns from these data is also skewed. This is a serious problem because the instances we want to learn about – the high-risk modules – are not being given priority by the learner.

Skewed datasets are not new to the machine learning community. The primary approaches to dealing with skewness are to resample the dataset, or to penalize the learner for not recognizing minority-class examples. In resampling, instead of simply taking the dataset as given, one can preprocess it so that the interesting cases make up a greater portion of the training data. This can be accomplished by undersampling the majority class, oversampling the minority class, or both. Alternatively, a learner can be penalized by modifying its global performance index to include a cost for each error. The cost per error can be higher for minority-class examples, and can thus force the learner to make fewer mistakes on minority-class examples. Interestingly, the only attempt to deal with skewness in the domain of software metrics has been to use differing misclassification penalties in a decision tree algorithm [29]. Our work applies a new resampling algorithm, SMOTE [9], to the problem of

skewness in metrics datasets. We then use the C4.5 decision tree learner to mine the resampled datasets. We compared the resulting trees against the trees generated from the original datasets, and found that they were much more accurate in identifying risky modules than the original trees.

The remainder of this paper is organized as follows. In Section 2, we will discuss the existing literature on the use of machine learning in software metrics, and how this effort fits in to the overall problem of software quality. In Section 3, we discuss our three datasets and the results of previous data mining experiments on them. In Section 4, we present our experimental results from the resampled datasets, and compare them to the original datasets. We offer a summary and discussion of future work in Section 5.

2. Software Metrics and Software Quality

In this section, we review the relevant literature from software quality management and machine learning. In particular, we look at the use and analysis of software metrics and approaches for overcoming skewness in a dataset.

2.1 Software Metrics

The Capability Maturity Model (CMM) is the de facto standard for rating how effective an organization's software development process is. The mode defines five levels of software process maturity: *Initial, Repeatable, Defined, Managed,* and *Optimizing*. The *Initial* level describes organizations that have no project management system at all. The *Repeatable* level refers to organizations that possess a rudimentary project management system that at least tracks software functionality, cost, and schedule. The *Defined* level describes organizations with complete, standardized project management and project engineering frameworks. The *Managed* level describes organizations that collect information about software quality and their development process, and use that information to improve their process. Finally, the *Optimizing* level refers to those few organizations that continually measure and

improve their development process, while also exploring process innovations [41]. According to a 2001 industry survey, more than a quarter (27.1%) of the organizations surveyed were at the *Initial* level, 39.1% were at the *Repeatable* level, and 23.4% were at the *Defined* level. Only 10.4% had reached the *Managed* or *Optimizing* levels [8]. An important fact to note is that the difference between the *Defined* and *Managed* levels is the use of product measurement, and its application to quality management.

Software metrics are the key tools in software quality management. Quality management is an ongoing comparison of the actual quality of a product with its expected quality. In the field of software development, this means that software metrics are collected at various points in the development cycle, and used to guide improvements to the quality of a software product [10],[19]. Metrics are used to identify modules in software systems that are potentially error-prone, so that extra development and maintenance effort can be directed at those modules. An oft-quoted rule is that 80% of a system's bugs will be found in just 20% of the system's modules [30]. Some of the statistical tools used in the analysis of software metrics include linear least-squares regression, robust regression, local polynomial regression, and M-estimation [21].

Each software metric quantifies some characteristic of a program. Simple counting metrics such as the number of lines of source code or Halstead's number of operators and operands [23], [25] simply describe how many "things" there are in a program. More complex metrics such as McCabe's cyclomatic complexity [37] or Belady's Bandwidth [3] attempt to describe the "complexity" of a program, by measuring the number of decisions in a module or the average level of nesting in the module, respectively. While different metrics do measure different characteristics, the various metrics tend to be strongly correlated to each other and to the number of failures in a program [18]. Furthermore, there tend to be relatively few modules in any given system that will have a high degree of complexity. As a result, any database of software characteristics or failures will be heavily skewed towards simple modules with a low occurrence of failures [2].

Not surprisingly, soft computing and machine learning techniques for modeling software failures have also been tried. Neural networks [26], [21], [31], [27], [48], neuro-fuzzy systems [1], [38], fuzzy logic and classification [18], [19], [22], genetic programming [30], genetic-fuzzy systems [2], fuzzy and rough sets [43], and classification trees [10], [28], [29] have been employed in various datasets. Multilayer perceptrons, in particular, are one of the more popular non-parametric techniques used in the analysis of software metric data. As mentioned, a constant problem for both statistical and soft computing approaches is that the data are heavily skewed in favor of modules with relatively few failures and relatively low metric values. This can be an especially serious problem for machine learning approaches that try to optimize a global measure of predictive accuracy; "always guessing the majority class" is a common mistake for learning algorithms in such skewed datasets. In [29], this issue of skewness was accounted for by the use of differing misclassification penalties in CART decision trees. A greater cost was associated with classifying high-risk modules as low risk than with classifying low risk modules as high risk. While this approach can improve the learning process in a skewed dataset, the results are highly sensitive to the ratio between the different misclassification penalties.

Our present work involves the use of a resampling algorithm named SMOTE (Synthetic Minority Oversampling TEchnique), which deals with skewness by both undersampling the majority classes and oversampling minority classes. The oversampling process also involves generating synthetic examples of the minority class, in order to expand the resulting decision regions. This latter feature is quite important; as can be seen in Tables 9, 12, and 14, the class distributions in our three datasets are quite badly skewed; the smallest class in OOSoft has only 5 examples associated with it. By generating synthetic examples, we can expand our ability to learn about these very small classes.

2.2 Machine Learning in Skewed Datasets

In many interesting machine learning problems, objects are not homogeneously distributed among the different classes. Very often, the

available data mostly consist of predominantly "normal" examples, with only a few "abnormal" examples. The abnormal examples, however, are precisely the ones that are most interesting. In addition, when there are costs to misclassifying an example, the cost of mistaking an "abnormal" example for a "normal" example is often much higher than classifying a "normal" example as "abnormal". The problem this poses is that a machine-learning algorithm usually works by defining some global performance index to rate the algorithm's current representation of a given problem. Learning then involves changing the problem representation to optimize that global index. This may involve adding new branches to a tree, updating connection weights in a neural network, producing a new generation of solutions in a genetic algorithm, adding or modifying rules in an expert system, etc. Clearly, when the "abnormal" cases are just a tiny fraction of the population, they cannot have a very large effect on the global index, and will thus be ignored to some extent by the learner.

The machine learning community has used several different approaches to overcome skewness in a dataset. The two most common are misclassification penalties and resampling techniques. Misclassification penalties are used to "punish" a learning algorithm when it makes an error. By associating a different penalty with different types of mistakes, the user can force a learner to avoid certain kinds of mistakes, at the cost of making more errors of a different type. Thus, mistaking an "abnormal" case for a "normal" one might carry a higher penalty than mistaking a "normal" case for an "abnormal" one, or vice versa. The precise penalty strategy depends on the problem domain and the user's goals. Misclassification penalties are an option in CART trees, and can be implemented for a variety of machine learning algorithms through post-processing [42], [14].

Resampling is the other major technique for dealing with skewed datasets. Resampling in this sense is distinct from bagging or boosting. Bagging is a resampling technique intended to create an ensemble of classifiers; a dataset is sampled *with replacement* to create a new dataset of the same size. This dataset is, properly speaking, not a set, since it can contain repeated elements. It is instead known as a *bag*. Plainly, the class

distribution in the bag will be roughly the same as in the original dataset. A collection of such bags is formed, and then an ensemble of classifiers is trained on the resulting bags, one classifier to each bag. The test set of inputs is then submitted to the ensemble, which votes on the final classification. The combination of bagging and voting is usually superior to creating a single classifier, provided that the learning algorithm is unstable [6]. Boosting is another technique for creating classifier ensembles. Boosting algorithms such as AdaBoost [47] sequentially train classifiers, placing more emphasis on certain patterns in each successive iteration. This is done by defining a probability density over the training data. For learning algorithms that do not support weighted training patterns, the same effect can be achieved by resampling the dataset with replacement, according to the desired probability density. This is the primary difference between boosting and bagging, since bagging uses *uniform* sampling with replacement.

The resampling techniques that interest us in the current paper are sometimes also referred to as *stratification*. They are used to alter the class distributions within a dataset, either to homogenize them or to make the classifier more sensitive to misclassification costs (as mentioned in [14]). The simplest approach is under-sampling, wherein a subset of majority-class examples are randomly selected for inclusion in the resampled dataset *without* replacement. This effectively thins out the majority class, making the dataset more homogeneous. Similarly, a simple over-sampling approach would be to duplicate examples from the minority class and include them in the resampled dataset [9]. More advanced techniques are also available in the literature; for instance, an under-sampling technique that preserves the class boundaries in a dataset is used in [32]. This is accomplished using the concept of Tomek links from statistical theory. In another vein, the SMOTE algorithm [9] creates synthetic examples in the minority class to be added to the genuine examples in the minority class. This is an over-sampling technique that was originally motivated by decision-tree learning. The authors found that simply replicating examples from the minority class causes decision trees to construct a small, tightly focused decision region around the replicated examples. As an alternative, the authors created synthetic

examples along the line in feature space that connects a minority class example to its nearest neighbor in the same class. They found that this approach resulted in an expanded decision region, and thus better generalization. A somewhat different application of under-sampling is the "uncertainty sampling" technique in [33]. The problem area in that paper is automatically labeling unlabelled examples in datasets; however, skewness also affects any effort to automatically categorize these unlabelled examples.

There are also other approaches to skewed datasets that do not fall under the umbrella of resampling or error penalties. Bayesian networks are often used to represent the probability structure of a dataset, but their performance as classifiers is sometimes suspect. In [20], the authors use classification accuracy as the driving goal in forming a Bayesian network. The resulting network performs well, but is considerably different from a traditional Bayesian network. By contrast, DeRouin and Brown [11] approach neural-network learning for skewed datasets by adding an adaptive *attention factor* to the learning rate of each neuron. The attention factor depends on the class distribution of the dataset, and on the proportion of each class that has already been presented for training.

In addition to methods for overcoming skewness, machine learning researchers have also been investigating the performance measures used to compare different algorithms. The traditional measure, classification error, has been extensively attacked for not incorporating the differing costs of different mistakes, and for not offering a complete picture of the relative performance of two classifiers. Provost and Fawcett [45] have instead argued for the use of the Receiver Operating Characteristic (ROC) curve from signal processing as a superior measure of classifier performance. Other authors have offered their own interpretations of the ROC curve [15], or used the curve itself to create new metrics for classifier performance [5]. A number of metrics are available for the specialized task of evaluating collections of text documents, where classes may be both skewed and sparse [16], [39]. An important point to note is that the ROC curve (and to a large extent the metrics for text search) only measure how well one class is learned. The ROC curve plots

the number of "True Positives" versus "False Positives", while the text-search metrics of precision and recall are based on the correct or incorrect placement of examples in a category. A metric that naturally measures a classifier's performance in several classes at once is the geometric mean, given by

$$P = \sqrt[k]{acc_1 \cdot acc_2 \cdot \ldots \cdot acc_k} \quad (1)$$

where acc_i is the classification accuracy for class i alone.

3. Previous Experimental Results

We have obtained three software metrics datasets, which were collected during the course of two Master's theses at the University of Wisconsin-Milwaukee [34], [36], [12]. In this section, we discuss the characteristics of these datasets, and the previous work that has been conducted on these datasets. In particular, we review the results of several fuzzy cluster analysis experiments reported in [13].

3.1 The MIS Dataset

This dataset consists of 390 records, each having 12 fields. The first 11 fields are the values of different software metrics for a module, and the final field is the number of changes made to that module, as measured by the number of problem reports on file. It is assumed that the number of changes corresponds to the number of failures in that module. The application that was analyzed is a commercial medical imaging system, General Electric's SIGNA system, running on a Data General MV4000 computer. In total, the system is comprised of approximately 400,000 lines of source code, divided into 4500 modules. Of these modules, 58% were written in Pascal, 29% were written in Fortran, 7% in assembly language, and 6% in PL/M (the Intel-86 programming language for microcomputers). The dataset itself was created by extracting the software metrics for a sample of 390 modules written in Fortran and Pascal, and associating those values with the number of changes that had to be made for each module [34].

The measures used in this data set are as follows:

(i) The number of lines of source code
(ii) The number of executable lines (rather than comments or white space)
(iii) The total number of characters
(iv) The number of comment lines
(v) The number of comment characters
(vi) The number of code characters
(vii) Halstead's N – defined as the number of operators plus the number of operands [23]
(viii) Halstead's N^ -- an approximation to N[23]
(ix) Jensen's NF – another approximation to N [25]
(x) McCabe's cyclomatic complexity [37]
(xi) Belady's Bandwidth [3]

An important point about this dataset is what constitutes a "module". Upon a first review, we noticed a large number of values that did not seem to make sense. In particular, non-integer values like "8.5" were reported for McCabe's cyclomatic complexity – a metric that should always consist of integer values. While this point is not specifically addressed in [34], it turns out that a "module" in this dataset is a source file, which may contain one or more routines. The values of the counting metrics, such as the lines of source code or the number of comment characters, have been determined by summing the values over all routines in a module. The values of the complexity metrics, such as Belady's Bandwidth or McCabe's cyclomatic complexity, have been determined by *averaging* over all routines in a module. Thus, the granularity of this dataset is fairly coarse. (As a side note, McCabe's seminal paper does account for modules composed of individual functions. The rule typically used to determine cyclomatic complexity, "number of decisions + 1", is actually the special case where cyclomatic complexity is determined for a single function. In [37], multiple functions are each treated as a strongly-connected component, and the total complexity is the sum of their individual complexities. Thus, in a module with multiple functions, it is actually better to sum the cyclomatic complexity values, rather than averaging them).

Lind's thesis reports on the result of a linear correlation analysis of this dataset. Pearson's correlation coefficient is computed between each metric and the change count. For the reader's convenience, we reproduce these results in Table 1. Notice that there is generally a strong positive correlation between each metric and the number of changes, except for Belady's Bandwidth.

Table 1. Correlation of Metrics to Changes, from [34]

Metric	Correlation to Changes
Total Lines	0.73
Code Lines	0.68
Total Chars	0.72
Comments	0.75
Comment Chars	0.66
Code Chars	0.69
Halstead's N	0.62
Halstead's N^	0.66
Jensen's NF	0.66
McCabe's	0.68
Belady's Band.	0.26

In addition to [34], Lind and Vairavan published a paper summarizing these results [35]. Another work that examines the MIS dataset is [40], in which the authors apply Principal Components Analysis as well. In this work, the number of principal dimensions was found to be 2, rather than 1 as in [13]. The difference is that in [40], the PCA algorithm was applied to the raw database, whereas in [13], the dataset was normalized first. Normalization is an important step, because differences in scale across different dimensions can distort the distribution of a multi-dimensional database. The PCA algorithm assumes a hyperellipsoidal distribution of data; distortions introduced by dimensional scalings can distort the true distribution, altering the orientation and size of the different axes.

Another work that examines this dataset is [26]. In this paper, the authors use neural networks for software reliability prediction and to identify fault-prone modules. The MIS dataset is used to illustrate the second objective. The 390 records are first classified into low-, medium-, and high-risk modules. The criterion used is that a low-risk module had no faults or 1 fault, medium-risk modules had 2-9 faults, and high-risk modules had 10 or more faults. The study considers only the 203 low- and high- risk modules; the medium-risk modules were discarded. The network was then trained to distinguish between low-risk and high-risk modules.

We would like to make two points concerning [26]. Firstly, the "hard" classification of modules based on the number of changes is a poor and arbitrary choice. No analysis has been conducted that showed that a module with 9 changes was substantially different than one with 10 changes. This artificial classification can dramatically *worsen* the performance of a neural network classifier. This is because the underlying assumption of a neural network is that the input/output pairings it is trained on represent *actual observations*, not subjective judgments. The network will *always* seek a smooth mapping from inputs to outputs, even when the introduction of subjective judgments has destroyed the actual mapping. Second, the results in [13] show that the fault classes that exist in the MIS dataset are in fact overlapping, *fuzzy* classes. The artificial imposition of hard boundaries, which are not truly representative of the data, will also distort a neural network classifier's results.

3.2 The OOSoft and ProcSoft Datasets

The two remaining datasets were collected by DeVilbiss in [12]. They contain records of software module characteristics, which are not associated with a change count. Both datasets are from operator display applications, which allow limited data entry. We have chosen to designate these datasets as the ProcSoft and OOSoft datasets. The application underlying the ProcSoft dataset was programmed using structured analysis techniques, in a mixture of C and assembly language.

The assembly language code was primarily for device drivers, and was ignored in [12]. The application underlying OOSoft was developed using object-oriented techniques. This program incorporates additional functionality over and above the functionality of the first program. In order to make a comparison, 422 functions from the structured program were analyzed, and 562 methods from the object-oriented program performing the same functionality were analyzed. Functions from the structured program, and methods from the object-oriented program, were treated as the basic modules of the program. Thus, these datasets represent a more fine-grained analysis than the MIS dataset.

The OOSoft dataset contains 562 records, and the ProcSoft dataset contains 422 records, with each record representing one method or function. There were a total of 11 measures computed for each function or method. These are:

(xii) n1 – The number of unique operators [23]
(xiii) n2 – The number of unique operands [23]
(xiv) N1 – The number of operators [23]
(xv) N2 – The number of operands[23]
(xvi) Halstead's N [23]
(xvii) Halstead's N^ [23]
(xviii) Jensen's NF [25]
(xix) VG1 – McCabe's cyclomatic complexity [37]
(xx) VG2 – McCabe's cyclomatic complexity, enhanced to include the number of predicates in decisions
(xxi) Lines of Code (LOC)
(xxii) Lines of Comments (CMT)

DeVilbiss examined the linear correlations between each pair of metrics in each dataset - excluding N, N^, and NF - again using Pearson's correlation coefficient. For the reader's convenience, we reproduce those results in Tables 2 & 3 below.

Note that in Table 2, all the metrics correlated well with each other, with the exception of the number of lines of comments. In Table 3, the correlation values tended to be lower. In that dataset, DeVilbiss identified a trend of similar metrics (such as the various metrics due to

Halstead [23]) being more correlated to each other than to different 'families' of metrics. Also, the number of lines of code and of comments correlated well only with each other.

Table 2. Pairwise Correlations in ProcSoft, from [12]

	n1	n2	N1	N2	VG1	VG2	LOC	CMT
n1	1.0	0.837	0.817	0.791	0.771	0.785	0.751	0.553
n2		1.0	0.933	0.948	0.864	0.857	0.806	0.554
N1			1.0	0.984	0.912	0.914	0.829	0.534
N2				1.0	0.879	0.877	0.820	0.531
VG1					1.0	0.982	0.760	0.476
VG2						1.0	0.755	0.475
LOC							1.0	0.908
CMT								1.0

Table 3. Pairwise Correlations in OOSoft, from [58]

	n1	n2	N1	N2	VG1	VG2	LOC	CMT
n1	1.0	0.814	0.796	0.681	0.636	0.597	0.193	-0.107
n2		1.0	0.869	0.883	0.678	0.670	0.318	-0.010
N1			1.0	0.952	0.882	0.864	0.447	0.072
N2				1.0	0.804	0.817	0.472	0.113
VG1					1.0	0.961	0.484	0.148
VG2						1.0	0.472	0.147
LOC							1.0	0.92
CMT								1.0

3.3 Fuzzy Cluster Analysis

Dick et al. [13] conducted a series of fuzzy clustering experiments on these three datasets. We will review selected results from that work that directly bear on our current paper. The three datasets contain no missing values, and were normalized to [0,1] for all attributes. Fuzzy clustering was carried out using the Fuzzy c-Means algorithm as implemented in MATLAB® 6.0. A fuzzifier value of 2.0 and a stopping criteria of 0.00001 were used. The Fuzzy c-Means algorithm does not select an optimal number of clusters; in fact, the number of clusters is an input to

the algorithm. In common with established practice, various numbers of clusters were tried, and cluster validity metrics were applied to each of the resulting fuzzy partitions [24]. The cluster validity metrics were the partition coefficient [4], the CS index [17], the Separation index [50], and (in the case of the MIS dataset) the average sum of squared error in a ten-fold cross-validation experiment. The results of these experiments are presented in Tables 4-6 for the MIS, OOSoft and ProcSoft datasets, respectively.

Table 4. Cluster Validity Measures for MIS

Clusters	Partition Coef	CS Index	Separation	Average SSE * 10^3
2	**0.8832**	**0.0064**	**9.5549**	3.3596
3	0.7275	0.0001	20.7707	2.7435
4	0.6529	0.0003	20.8820	2.7124
5	0.6031	0.0000	16.5433	2.5807
6	0.5145	0.0005	29.1250	2.5836
7	0.4865	0.0006	23.9647	**2.5753**
8	0.4262	0.0001	41.6580	2.5878
9	0.4060	0.0001	40.0919	2.5762
10	0.4014	0.0009	26.0038	2.5821

When we examine each table to determine the optimal number of clusters, we find that all three tables contain disagreements between the different metrics. Maximum values of the partition coefficient and the CS index indicate optimal partitions, while minimal values of the Separation index and the cross-validation error are best. In Table 4, the partition coefficient, CS index, and Separation index all indicate that 2 clusters is the optimal number – but the tenfold cross-validation error indicates that 7 clusters is the best choice. In Table 5, both the CS index and the Separation index indicate that 5 clusters are best, in contrast to the partition coefficient, which indicates that 2 clusters is the optimal number. Likewise, in Table 6, the partition coefficient and the CS index both point to 2 clusters as the best partition, while the Separation index indicates that 8 clusters are best. After careful examination of the results, we have chosen the following partitions: 7 clusters in MIS, and 5 clusters

in OOSoft. Our rationale for these choices is that there is a common behavior at each of those values. In MIS, the tenfold cross-validation error is the best metric, as it is based on the true output values in this dataset. In addition, both the Separation index and the CS index have local extrema at 7 clusters, as can be seen in Figure 1. In the OOSoft dataset, the common indication from the Separation and CS indices is too strong to ignore; these are generally considered superior to the partition coefficient.

Table 5. Cluster Validity Metrics for OOSoft

Clusters	Partition Coef	CS Index	Separation Index
2	**0.7371**	$1.0164 * 10^{-4}$	229.2901
3	0.6601	$3.1147 * 10^{-4}$	184.2069
4	0.6704	$3.1147 * 10^{-4}$	95.8886
5	0.6695	$\mathbf{6.9479 * 10^{-4}}$	**8.9305**
6	0.5938	$4.9494 * 10^{-4}$	16.0950
7	0.5383	$4.1993 * 10^{-4}$	17.9859
8	0.5150	$1.0724 * 10^{-4}$	17.2931
9	0.5037	$6.4591 * 10^{-4}$	16.1833
10	0.4767	$6.4591 * 10^{-4}$	17.8566

Table 6. Cluster Validity Metrics for ProcSoft

Clusters	Partition Coef	CS Index	Separation Index
2	**0.8795**	**0.0012**	20.0554
3	0.7524	$1.4633 * 10^{-4}$	39.6065
4	0.6951	$3.4166 * 10^{-4}$	21.4158
5	0.6138	$4.2281 * 10^{-5}$	33.1550
6	0.5450	$1.5024 * 10^{-4}$	35.7366
7	0.5392	$2.3218 * 10^{-4}$	13.5172
8	0.5070	$1.1190 * 10^{-4}$	**12.8757**
9	0.4738	$1.7293 * 10^{-4}$	13.0638
10	0.4416	$3.1477 * 10^{-4}$	19.4462

The ProcSoft dataset presents a somewhat more challenging problem. The Separation index has a local minimum at 2 clusters, and we would be inclined to select 2 clusters as the correct partition based on the cluster

validity metrics alone. However, the resulting partition is skewed towards large values; as can be seen in Table 7, the cluster of smaller values (cluster 1) is much smaller than the cluster of larger values (cluster 2). This is diametrically opposed to the true situation in this dataset; by examining the mean and median values in Table 8, we see that the dataset is in fact strongly skewed towards smaller values. Thus, we are forced to conclude that 8 clusters is the correct partition for this dataset.

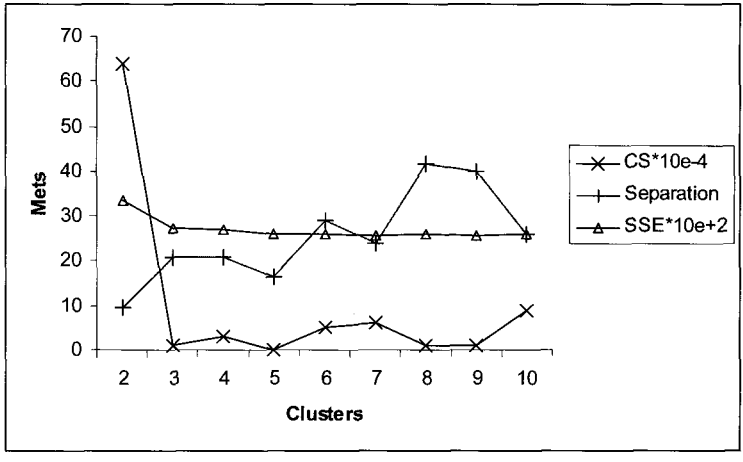


Figure 1. Cluster Validity Metrics for MIS [13]

Table 7. Number of Examples in ProcSoft

Cluster ID	Count
1	59
2	363

Table 9 is a statistical characterization of the changes per module in each cluster of the MIS dataset. For each cluster, the minimum, maximum, mean, median, standard deviation and centroid of the number of changes per module, along with the count of examples in each cluster,

have been computed. The clusters appear to follow a linear pattern; in all 11 input dimensions, the cluster centers are monotonically increasing, as can be seen in Table 10. This means that, for each cluster, the values of each metric are monotonic increasing with respect to the other clusters, and this pattern holds across every metric dimension. Furthermore, the mean, median, standard deviation and centroid of the changes per module are monotonic increasing for this ordering, while the number of examples in the cluster is monotonic decreasing for this ordering.

Table 8. Statistical Characterization of the ProcSoft dataset

	MIN	MAX	MEAN	MEDIAN	STD
n1	1.00	54.00	16.48	15.00	9.52
n2	1.00	151.00	17.91	12.00	18.94
N1	1.00	884.00	75.28	37.00	110.17
N2	1.00	498.00	44.80	20.50	68.60
N	2.00	1303.00	120.08	56.50	178.10
NH	1.00	1292.00	155.78	101.50	170.66
NF	0.00	1029.00	112.48	69.50	132.47
McCabe1	1.00	67.00	5.39	3.00	6.87
McCabe2	1.00	72.00	6.12	3.00	7.87
LOC	3.00	786.00	84.92	55.00	92.83
CMT	0	524	51.06872	34	60.26566

Table 9. Cluster Characteristics in MIS

Cluster	**Min**	**Max**	**Mean**	**Median**	**STD**	**Count**	**Centroid**
1	0	27	4.18	2	4.68	102	4.75
2	0	47	21.25	16.5	12.94	20	21.15
3	8	41	19.32	14	12.12	22	16.56
4	0	19	2.26	1	3.10	107	3.12
5	14	98	36.75	32.5	22.00	12	29.06
6	0	25	5.33	4	4.89	86	6.16
7	1	46	10.02	7	9.38	41	9.92

Table 10. Ordering of Cluster Centers for Each Attribute

Attribute	Ordering of Clusters
A: Lines of Source Code	4, 1, 6, 7, 3, 2, 5
B: Executable Lines	4, 1, 6, 7, 3, 2, 5
C: Total Characters	4, 1, 6, 7, 3, 2, 5
D: Comment Lines	4, 1, 6, 7, 3, 2, 5
E: Comment Characters	4, 1, 6, 7, 3, 2, 5
F: Code Characters	4, 1, 6, 7, 3, 2, 5
G: Halstead's N	4, 1, 6, 7, 3, 2, 5
H: Halstead's N^	4, 1, 6, 7, 3, 2, 5
I: Jensen's NF	4, 1, 6, 7, 3, 2, 5
J: McCabe's Cyclomatic Complexity	4, 1, 6, 7, 3, 2, 5
K: Belady's Bandwidth	4, 1, 6, 7, 3, 2, 5
Output: Change Centroids	4, 1, 6, 7, 3, 2, 5

Table 11. Ordering of Cluster Centers in OOSoft

Feature	Cluster Center Ordering
n1	1, 3, 2, 4, 5
n2	1, 3, 2, 4, 5
N1	1, 3, 2, 4, 5
N2	1, 3, 2, 4, 5
N	1, 3, 2, 4, 5
Nh	1, 3, 2, 4, 5
Nf	1, 3, 2, 4, 5
McCabe1	1, 3, 2, 4, 5
McCabe2	1, 3, 2, 4, 5
LOC	1, 2, 4, 3, 5
Comments	1, 2, 4, 5, 3

Some important observations can be drawn from Table 9. Firstly, the dataset is indeed deeply skewed; the clusters are skewed towards smaller metric values and smaller change rates, while each individual cluster is itself skewed in this direction, as can be observed from the difference between mean and median values. Secondly, there appears to be a phenomenon of increasing variance with increasing change rates. This can be found in the monotonic relationship between the means and standard deviations. We can thus make the following statement: in the

MIS datatset, the most interesting classes (the ones with the highest change rates) tend to have few examples and a markedly higher variance than non-interesting classes. This combination of characteristics will be quite challenging for machine learning algorithms, and especially function approximation algorithms. In fact, this experimental evidence indicates that function approximation would be much less useful than a classification approach to data mining in this dataset. It is the nature of classification approaches that the variance present in the output dimension is reduced, when compared to function approximation approaches. The aggregation of multiple examples into a single class simplifies the class recognition problem.

Table 12. Number of Examples in OOSoft

Cluster ID	**Count**
1	195
3	60
2	207
4	95
5	5

Table 13. Ordering of Cluster Centers in ProcSoft

Attributes	**Ordering - 8 Clusters**
n1	2, 5, 7, 4, 6, 8, 3, 1
n2	2, 5, 7, 4, 6, 8, 3, 1
N1	2, 5, 7, 4, 6, 8, 3, 1
N2	2, 5, 7, 4, 6, 8, 3, 1
N	2, 5, 7, 4, 6, 8, 3, 1
N^	2, 5, 7, 4, 6, 8, 3, 1
NF	2, 5, 7, 4, 6, 8, 3, 1
McCabe1	2, 5, 7, 4, 6, 8, 3, 1
McCabe2	2, 5, 7, 4, 6, 8, 3, 1
Lines of Code	2, 5, 7, 4, 6, 8, 3, 1
Comments	2, 5, 7, 4, 6, 8, 3, 1

The OOSoft and ProcSoft datasets do not include change data, so our characterization is limited to the ordering of cluster centers and the number of examples per cluster. In Tables 11 & 12, the ordering of cluster centers for OOSoft and the number of examples per cluster are given. Interestingly, there is some disagreement in the ordering in different dimensions, and the number of examples per cluster is not monotonically increasing or decreasing for any of these orderings. The ordering of cluster centers in ProcSoft and the number of examples per cluster are given in Tables 13 & 14, respectively.

Table 14. Number of Examples in ProcSoft

Cluster ID	Count
1	3
2	82
3	11
4	59
5	136
6	23
7	83
8	25

4. Experimental Results

We have conducted a series of data mining experiments in these three datasets using the C4.5 decision tree generator, release 8. Each experiment was a tenfold cross-validation, with the examples in each partition being selected by a stratified sampling algorithm. Thus, each individual partition had the same class distribution as the original dataset. The classes used were the fuzzy clusters discussed in Section 3.3, hardened into classes using the method of maximum membership [24]. For each class of interest, we compute the class testing accuracy by dividing the number of correct examples through all ten runs by the total number of examples in that class. The overall performance in each experiment was determined by taking the geometric mean of the accuracies in the classes of interest, given by

$$P = \sqrt[k]{acc_1 \cdot acc_2 \cdot \ldots \cdot acc_k} \quad (2)$$

for the k classes of interest in the dataset. Perfect accuracy for all classes of interest will be reflected by a value of $P = 1$.

Resampling approaches work by identifying one or more classes as being interesting, and then altering the distribution of the dataset to favor those classes, at the expense of uninteresting classes. A resampling strategy that improves the learning of one class will degrade the representation of other classes. Thus, our first task is to identifying those classes that are of interest in our three datasets. For the MIS dataset, we turn back to Table 9 and look at the mean number of changes per module in each class. We make the (admittedly arbitrary) decision that we want to identify those modules that belong to any class having an average of more than 10 changes. This yields 4 classes of interest: clusters 2, 3, 5 and 7. For the OOSoft and ProcSoft datasets, change counts are unavailable. Based on the known fact that higher metric values correlate well with higher failure rates, we want to identify classes in these two datasets that have unusually high metric values. We do this by comparing the cluster centers in each dimension with the mean value for the whole dataset in that dimension. The classes of interest in ProcSoft and OOSoft are those clusters whose centers are higher than the mean value for each dimension in at least two dimensions.

Table 15. Tenfold Cross-Validation Results for MIS

	10-Fold Accuracy
Overall	84.9%
Class 1	80.39%
Class 2	80%
Class 3	77.27%
Class 4	93.46%
Class 5	91.67%
Class 6	88.37%
Class 7	75.61%
Performance	0.8091

We began by experimenting with the original datasets. The overall accuracy, class accuracy, and geometric mean for the MIS, OOSoft and ProcSoft datasets are given in Tables 15, 16, and 17. The classes of interest in the MIS dataset were classes 2, 3, 5, and 7. In OOSoft, the classes of interest are classes 3, 4, 5, and in ProcSoft they were classes 1, 3, 8. We found, in general, that the classes with the highest accuracy were the ones that either had the lowest or highest metric values; those classes in between were more difficult.

Table 16. Tenfold Cross-Validation Results for OOsoft

	10-Fold Accuracy
Overall	97.7%
Class 1	98.97%
Class 2	98.02%
Class 3	96.9%
Class 4	94.89%
Class 5	100%
Performance	0.9724

Table 17. Tenfold Cross-Validation Results for ProcSoft

	10-Fold Accuracy
Overall	89.1%
Class 1	66.67%
Class 2	95.12%
Class 3	81.82%
Class 4	79.66%
Class 5	95.59%
Class 6	78.26%
Class 7	87.95%
Class 8	76%
Performance	0.7456

Following these initial experiments, we used undersampling and SMOTE to alter the class distributions in each dataset. Our goal was to

trade off decreased accuracy in uninteresting classes for increased accuracy in the ones that were interesting. We are thus calibrating a decision tree to perform best on the classes that are of interest to us. In general, we undersampled uninteresting classes to as little as 25% of their original population, and oversampled interesting classes by 100 or 200%. In the terminology of our paper, X% undersampling means we create a class with X% of the number of examples in the original class, chosen through random uniform sampling without replacement. Y% oversampling means that we create a class that contains Y/100 times as many synthetic examples as there were examples in the original class, *plus* all of the original examples. The experiments followed an iterative, exploratory process, and we stopped when all of the interesting classes had a class accuracy greater than the overall accuracy for the entire dataset.

Table 18. Resampling Strategies in MIS

Class	Experiment 1	Experiment 2	Experiment 3	Experiment 4
1	50% undersample	50% undersample	25% undersample	25% undersample
2	100% oversample	200% oversample	100% oversample	200% oversample
3	100% oversample	200% oversample	200% oversample	200% oversample
4	50% undersample	50% undersample	25% undersample	25% undersample
5	Unchanged	Unchanged	Unchanged	100% oversample
6	50% undersample	50% undersample	50% undersample	50% undersample
7	100% oversample	100% oversample	100% oversample	100% oversample

Table 19. Resampling Results in MIS

	Experiment 1	Experiment 2	Experiment 3	Experiment 4
Overall	90.4%	88.9%	89.8%	89%
Class 1	88.24%	84.31%	80.77%	50%
Class 2	90.0%	88.33%	85%	96.67%
Class 3	88.64%	93.94%	98.48%	96.97%
Class 4	94.4%	87.04%	92.59%	74.07%
Class 5	100%	91.67%	91.67%	95.83%
Class 6	81.4%	79.07%	69.77%	81.4%
Class 7	93.9%	92.68%	96.34%	96.34%
Index	0.9303	0.9163	0.9273	0.9645

Table 20. Resampling Strategy for OOSoft

	Experiment 1	Experiment 2	Experiment 3
Class 1	50% undersample	50% undersample	25% undersample
Class 2	50% undersample	50% undersample	25% undersample
Class 3	Unchanged	100% oversample	100% oversample
Class 4	Unchanged	Unchanged	100% oversample
Class 5	Unchanged	Unchanged	Unchanged

For the MIS dataset, we conducted 4 resampling experiments. The resampling strategy for each of these experiments is shown in Table 18. Table 19 presents the overall accuracy, class accuracies, and performance for each of these experiments, again determined through a tenfold cross-validation experiment using C4.5. As can be seen, all of the resampling strategies resulted in an improved overall accuracy. We can also observe an improvement in the performance measure as we decrease the sampling rate for uninteresting classes and increase it for interesting cases. The procedure is sensitive to the exact combination of undersampling and oversampling used in the dataset. The optimal strategy appears to be specific to each dataset.

Table 21. Resampling Results for OOSoft

	Experiment 1	Experiment 2	Experiment 3
Overall	96.1%	97.2%	98.3%
Class 1	96.94%	98.98%	97.96%
Class 2	96.15%	94.23%	94.23%
Class 3	86.67%	99.17%	100%
Class 4	94.74%	95.79%	98.42%
Class 5	100%	100%	100%
Index	0.9364	0.9830	0.9947

For the OOSoft dataset, we conducted 3 resampling experiments. The sampling strategy for each experiment is described in Table 20, while the results of each experiment are presented in Table 21. Similarly, our resampling strategy for the 3 experiments conducted in the ProcSoft dataset is presented in Table 22, and the results of those experiments are presented in Table 23. As can be seen, the combination of undersampling and SMOTE is consistently able to alter the class accuracies to favor the

interesting classes, thereby calibrating a decision tree to find the modules in a specific project that have a higher likelihood of failure.

Table 22. Resampling Strategy for ProcSoft

	Experiment 1	Experiment 2	Experiment 3
Class 1	100% oversample	100% oversample	100% oversample
Class 2	50% undersample	50% undersample	25% undersample
Class 3	100% oversample	200% oversample	200% oversample
Class 4	Unchanged	Unchanged	Unchanged
Class 5	50% undersample	25% undersample	25% undersample
Class 6	Unchanged	Unchanged	Unchanged
Class 7	50% undersample	50% undersample	50% undersample
Class 8	100% oversample	100% oversample	100% oversample

Table 23. Resampling Results for ProcSoft

	Experiment 1	Experiment 2	Experiment 3
Overall	87.8%	87.2%	89.9%
Class 1	100%	100%	100%
Class 2	87.8%	95.12%	90.48%
Class 3	86.36%	93.94%	96.67%
Class 4	86.44%	83.05%	88.14%
Class 5	91.18%	88.23%	97.06%
Class 6	69.57%	73.91%	78.26%
Class 7	85.71%	83.33%	76.19%
Class 8	94%	88%	92%
Index	0.9329	0.9385	0.9407

5. Proposed Usage

We believe that a classifier, which has been calibrated to identify troublesome modules, will be of great benefit to software developers. Imagine the following scenario: a programmer completes a module for a software system, and checks it in to the configuration control system. A few minutes later, he receives an email telling him that the module he has checked in appears to be in the "moderately risky" category, meaning that there is a higher-than-normal risk of failure in that module, based on

the software metrics computed for that module. The programmer then has the opportunity to redesign the module to reduce its complexity, or to prepare a more rigorous testing plan. It is even possible that the project manager might set guidelines for how much testing is needed for each failure risk class. This could be based on a cost optimization model that accounts for the differing levels of risk associated with each risk category.

The above scenario is the ideal that researchers have been striving for in software metrics research. However, one of the key stumbling blocks is that we cannot tell *a priori* what metric values correspond to low risk, medium risk, or high risk categories. Take McCabe's cyclomatic complexity: one source [49] asserts that a cyclomatic complexity of more than 10 is associated with an increased failure rate, while another [2] might say 15. The real problem is that each development project is in large measure unique. Different team members bring a different set of skills and experience to bear on problems that might be in completely different application areas, and more or less difficult than each other. Notice that even calibrating metrics thresholds from a company's historical data is problematic, since the development team, application domain, and project difficulty are most likely different.

The solution we advocate in this paper is to return to an idea proposed by Brooks more than 30 years ago: building a pilot system for each project, with the intent to learn from the pilot system and then *throw it away* [7]. The software development community, for obvious reasons, has not embraced this idea; building a realistic pilot system will be expensive in itself, and thus would significantly increase the cost of software development. However, there are a number of results and observations gathered in the course of decades of software research that point to the usefulness of a pilot system:

- Brooks' argument that software development is a learning process certainly rings true. Developers constantly have to learn new application domains to produce products their customers want, application domains that none of the developers may have *any* understanding of. This gap between the domain of software development and the application domain is extremely dangerous. A

pilot system will give developers a chance to develop competence in the application domain, before building the production system.

- In his seminal paper describing the waterfall model [46], W. W. Royce *also* argues for the development of a pilot system. His rationale is that matters such as timing and storage allocation should be explored through this pilot system. The developers will be able to experiment with a working system rather than relying on human judgement, which Royce characterizes as "invariably and seriously optimistic" in the area of software development. While timing and storage allocation are no longer the key issues they were in 1970, the underlying principle is the same: design decisions will be sounder if they arise from experimental studies rather than human judgement.
- The research on iterative development has showed that rapid prototyping helps determine a user's true requirements, and can lead to better software. Iterative development is now accepted as being far superior to a single, monolithic, design-and-build model. Building a pilot system would certainly fit into the iterative development model.
- Other engineering disciplines routinely build pilot systems, and accept the costs as part of the development cycle. No chemical engineer would build a production plant without first building and testing a pilot plant, while manufacturers routinely experiment with process designs before settling on one production process.
- Finally, [13] indicates that failure classes are best determined through a supervised learning algorithm. While the authors were able to cluster the OOSoft and ProcSoft datasets, and used these clusters to build a decision tree classifier, in the MIS dataset we observe that the cluster partition that best represented the actual occurrence of changes was *not* seen as optimal by any of the commonly used cluster validity metrics. Unsupervised learning is attractive because we do not have to wait until the system is implemented and tested before mining can begin; however, what we see in [13] is evidence that supervised learning is more accurate in this domain. If we need to use actual failure counts, then only two choices are available: either wait until failure counts become available late in the project

(which renders the metrics-based screening process moot), or gather failure data from a pilot system.

There is a general consensus in the software engineering community that we do not fully understand how to produce high-quality software. We have learned that the traditional "waterfall" model of software development is a poor fit to the special characteristics of software development [44], and that some form of iterative process is needed instead. Our contention is that *there is now a substantial body of evidence that favors building a pilot system as a routine step in software development.* In addition to its other benefits, a pilot system can also be used to calibrate software metrics to the project under development, enabling the construction of automated screening tools for software modules.

6. Conclusions

We have presented the first-ever application of resampling techniques in the domain of software metrics. We have found that resampling is indeed a viable technique for calibrating a decision tree to identify classes of interest in a database of software metrics. We have proposed that decision-tree calibration can best be carried out through the construction of a pilot system, which we believe would lead to higher-quality software in general. Our future work in this area will involve an examination of various data-mining algorithms, to determine if our resampling results can be generalized to any such algorithm, not just decision trees. We will also attempt to determine if oversampling or undersampling are themselves sufficient, or if both techniques are required to recalibrate a data-mining algorithm to the degree we have observed in this work.

Acknowledgement

This work was supported in part by the National Institute for Systems Test and Productivity under the USA Space and Naval Warfare Systems Command grant no. N00039-01-1-2248. Our thanks to Dr. Vairavan of

the University of Wisconsin-Milwaukee for providing the datasets used in this paper.

References

[1] Baisch, E.; Bleile, T.; Belschner, R., "A neural fuzzy system to evaluate software development productivity", in *Proceedings of the 1995 IEEE International Conference on Systems, Man and Cybernetics,* 1995, pp. 4603-4608.

[2] Baisch, E.; Liedtke, T., "Comparison of conventional approaches and soft-computing approaches for software quality prediction," in *Proceedings of the 1997 IEEE International Conference on Systems, Man and Cybernetics*, 1997, pp. 1045-1049

[3] Belady, L.A., "Software Geometry", *Proceedings of the International Computer Symposium*, Taipei, R.O.C., December 1980.

[4] Bezdek, J.C., *Pattern Recognition with Fuzzy Objective Function Algorithms*, New York: Plenum Press, 1981.

[5] Bradley, A.P., "The use of the area under the ROC curve in the evaluation of machine learning algorithms", *Pattern Recognition*, vol. 30 no. 7, 1997, pp. 1145-1159.

[6] Breiman, L., "Bagging Predictors", *Technical Report No. 421*, September 1994, Department of Statistics, University of California at Berkeley.

[7] Brooks, F.P., Jr., *The Mythical Man-Month:Essays on Software Engineering, Anniversary Edition,* Reading, MA: Addison-Wesley Pub. Co., 1995.

[8] Carnegie Mellon University, "SEMA Maturity Profile", http://www.sei.cmu.edu/sema/profile.html, August 24, 2001.

[9] Chawla, N.V.; Bowyer, K.W.; Hall, L.O.; Kegelmeyer, W.P., "SMOTE: synthetic minority over-sampling technique", *Journal of Artificial Intelligence Research*, vol. 16, June 2002, pp. 321-357.

[10] De Almeida, M.A.; Lounis, H.; Melo, W.L., "An investigation on the use of machine learned models for estimating software correctability", *International Journal of Software Engineering and Knowledge Engineering*, vol. 9 no. 5, 1999, pp. 565-593.

[11] DeRouin, E.; Brown, J., "Neural network training on unequally represented classes", in *Proceedings, Artificial Neural Networks in Engineering Conference*, 1991, pp. 135-140.

[12] DeVilbiss, W., "A Comparison of Software Complexity of Programs Developed Using Structured Techniques and Object-Oriented Techniques", Master's Thesis, University of Wisconsin-Milwaukee, 1993.

[13] Dick, S.; Meeks, M.; Last, M.; Bunke, H.; Kandel, A., "Data mining in software metrics databases", submitted to *Fuzzy Sets & Systems,* November 2001.

[14] Domingos, P., "MetaCost: a general method for making classifiers cost-sensitive", in *Proceedings, KDD-99,* San Diego, CA, USA, Aug. 15-18, 1999, pp. 155-164.

[15] Drummond, C.; Holte, R.C., "Explicitly representing expected cost: an alternative to ROC representation", in *Proceedings, KDD-2000*, Boston, MA, USA, August 20-23, 2000, pp. 198-207.

[16] Dumais, S.; Platt, J.; Heckerman, D., "Inductive learning algorithms and representations for text categorization", in *Proceedings, 1998 ACM Int. Conf. On Information and Knowledge Management*, Bethesda, MD, USA, November 3-7, 1998, pp. 148-155.

[17] Dunn, J.C., "A Fuzzy Relative of the ISODATA Process and its Use in Detecting Compact Well-Separated Clusters", *Journal of Cybernetics*, vol. 3 no. 3, 1973, pp. 32-57. Reprinted in Bezdek, J.C.; Pal, S.K., *Fuzzy Models for Pattern Recognition: Methods that Search for Structures in Data*, Piscataway, NJ: IEEE Press, 1992, pp. 82-101.

[18] Ebert, C., "Fuzzy classification for software criticality analysis", *Expert Systems with Applications*, vol. 11 no. 3, pp. 323-342, 1996.

[19] Ebert, C.; Baisch, E., "Knowledge-based techniques for software quality management", in W. Pedrycz, J.F. Peters, Eds., *Computational Intelligence in Software Engineering*, River Edge, NJ: World Scientific, 1998, pp. 295-320

[20] Ezawa, K.J.; Singh, M.; Norton, S.W., "Learning goal oriented Bayesian networks for telecommunications risk management", in *Proceedings, 13th Int. Conf. On Machine Learning*, Bari, Italy, July 3-6, 1996, pp. 139-147.

[21] Gray, A.R., "A simulation-based comparison of empirical modeling techniques for software metric models of development effort", in *Proceedings of the 6th International Conference on Neural Information Processing*, 1999, pp. 526-531.

[22] Gray, A; MacDonell, S., "Applications of Fuzzy Logic to Software Metric Models for Development Effort Estimation", *Proceedings of the Annual Meeting of the North American Fuzzy Information Processing Society -- NAFIPS*, Syracuse, NY, USA, September 21-24, 1997, pp. 394-399.

[23] Halstead, M., *Elements of Software Science*, New York: Elsevier, 1977.

[24] Hoppner, F.; Klawonn, F.; Kruse, R.; Runkler, T., *Fuzzy Cluster Analysis: Methods for Classification, Data Analysis and Image Recognition*, New York: John Wiley & Sons, Inc., 1999.

[25] Jensen, H.A.; Vairavan, K., "An Experimental Study of Software Metrics for Real-Time Software", *IEEE Transactions on Software Engineering*, vol. 11, no. 2, Feb. 1985, pp. 231-234.

[26] Karunanithi, N.; Malaiya, Y.K. "Neural Networks for Software Reliability Engineering", in M.R. Lyu, Ed., *Handbook of Software Reliability Engineering*, New York: McGraw-Hill, 1996.

[27] Khoshgoftaar, T.M.; Allen, E.B., "Neural networks for software quality prediction", in W. Pedrycz, J.F. Peters, Eds., *Computational Intelligence in Software Engineering*, River Edge, NJ: World Scientific, 1998, pp. 33-63

[28] Khoshgoftaar, T.M.; Allen, E.B.; Jones, W.D.; Hudepohl, J.P., "Classification-tree models of software-quality over multiple releases", *IEEE Transactions on Reliability*, vol. 49 no. 1, March 2000, pp. 4-11

[29] Khoshgoftaar, T.M.; Allen, E.B.; Jones, W.D.; Hudepohl, J.P., "Data Mining for Predictors of Software Quality", *International Journal of Software Engineering and Knowledge Engineering*, vol. 9 no. 5, 1999, pp. 547-563.

[30] Khoshgoftaar, T.M.; Evett, M.P.; Allen, E.B.; Chien, P.-D., "An application of genetic programming to software quality prediction", in W. Pedrycz, J.F. Peters, Eds., *Computational Intelligence in Software Engineering,* River Edge, NJ: World Scientific, 1998, pp. 176-195.

[31] Khoshgoftaar, T.M.; Szabo, R.M., "Using Neural Networks to Predict Software Faults During Testing", *IEEE Transactions on Reliability*, vol. 45 no. 3, Sept. 1996, pp. 456-462

[32] Kubat, M.; Matwin, S., "Addressing the curse of imbalanced training sets: one-sided selection", in *Proceedings, 14th Int. Conf. On Machine Learning,* Nashville, TN, USA, July 8-12, 1997, pp. 179-186.

[33] Lewis, D.D.; Catlett, J., "Heterogeneous uncertainty sampling for supervised learning", in *Proceedings 11th Int. Conf. On Machine Learning,* Rutgers University, New Brunswick, NJ, USA, July 10-13, 1994, pp. 148-156.

[34] Lind, R.K,"An Experimental Study of Software Metrics and Their Relationship to Software Errors", Master's Thesis, University of Wisconsin-Milwaukee, 1986.

[35] Lind, R.K.; Vairavan, K., "An Experimental Investigation of Software Metrics and Their Relationship to Software Development Effort", *IEEE Transactions on Software Engineering*, vol. 15 no. 5, May 1989, pp. 649-653.

[36] Lyu, M.R., "Data and Tool Disk", in M.R. Lyu, Ed., *Handbook of Software Reliability Engineering*, New York: McGraw-Hill, 1996.

[37] McCabe, T.J., "A Complexity Measure", *IEEE Transactions on Software Engineering*, vol. 2 no. 4, Dec. 1976, pp. 308-320.

[38] Mertoguno, J.S.; Paul, R,; Bourbakis, N.G.; Ramamoorthy, C.V., "A Neuro-Expert System for the Prediction of Software Metrics", *Engineering Applications of Artificial Intelligence*, vol. 9 no. 2, 1996, pp. 153-161.

[39] Mladenic, D.; Grobelnik, M., "Feature selection for unbalanced class distribution and naïve Bayes", in *Proceedings, 16th Int. Conf. On Machine Learning,* Bled, Slovenia, June 27-30, 1999, pp. 258-267.

[40] Munson, J.C.; Khoshgoftaar, T.M., "Software Metrics for Reliability Assessment", in M.R. Lyu, Ed., *Handbook of Software Reliability Engineering*, New York: McGraw-Hill, 1996.

[41] M.C. Paulk, B. Curtis, M.B. Chrissis, C.V. Weber, "Capability Maturity Model, Version 1.1", *IEEE Software*, vol. 10 no. 4, July 1993, pp. 18-27

[42] Pazzani, M.; Merz, C.; Murphy, P.; Ali, K.; Hume, T.; brunk, C., "Reducing misclassification costs", in *Proceedings, 11th Int. Conf. On Machine Learning,* Rutgers University, New Brunswick, NJ, USA, July 10-13, 1994, pp. 217-225.

[43] Pedrycz, W.; Peters, J.F.; Ramanna, S., "Design of a software quality decision system: a computational intelligence approach", in *Proceedings, IEEE Canadian Conference on Electrical and Computer Engineering*, Waterloo, ON, Canada, May 24-28, 1998, pp. 513-516.

[44] Peters, J.F.; Pedrycz, W., *Software Engineering: An Engineering Approach*, New York: John Wiley & Sons, Inc., 2000.

[45] Provost, F.; Fawcett, T.; Kohavi, R., "The case against accuracy estimation for comparing induction algorithms", in *Proceedings, 15th Int. Conf. On Machine Learning,* Madison, WI, USA, July 24-27, 1998, pp. 445-453.

[46] Royce, W.W., "Managing the development of large software systems", in *Proceedings, IEEE WESCON*, August 1970, pp. 1-9. Reprinted in *Proceedings, 9th International Conference on Software Engineering*, Monterey, CA, USA, March 30-April 2, 1987, pp. 328-338.

[47] Schwenk, H.; Bengio, Y., "AdaBoosting neural networks: application to on-line character recognition", in *Proceedings of ICANN '97*, Lausanne, Switzerland, Oct. 8-10, 1997, pp. 967-972.

[48] Shin, M.; Goel, A.L., "Knowledge Discovery and Validation in Software Metrics Databases", *Proceedings of SPIE - The International Society for Optical Engineering*, vol. 3695, April 1999, pp. 226-233.

[49] Stubbs, D.F.; Webre, N.W., *Data Structures with Abstract Data Types and Ada*, Boston, MA: PWS Pub. Co., 1993.

[50] Xie, X.L.; Beni, G., "A Validity Measure for Fuzzy Clustering", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 13 no. 8, August 1991, 841-847. Reprinted in Bezdek, J.C.; Pal, S.K., *Fuzzy Models for Pattern Recognition: Methods that Search for Structures in Data*, Piscataway, NJ: IEEE Press, 1992, pp. 219-225.

Artificial Intelligence Methods *in* Software Testing

An inadequate infrastructure for software testing is causing major losses to the world economy. The characteristics of software quality problems are quite similar to other tasks successfully tackled by artificial intelligence techniques. The aims of this book are to present state-of-the-art applications of artificial intelligence and data mining methods to quality assurance of complex software systems, and to encourage further research in this important and challenging area.

Key Features

- Coverage of novel methods for software testing and software quality assurance
- Introduction to state-of-the-art data mining models and techniques
- Analyses of new and promising application domains of artificial intelligence and data mining in software quality engineering
- Contributions from leading authors in the fields of software engineering and data mining

www.worldscientific.com
5549 hc

ISBN 981-238-854-0
9 789812 388544